VISHWA BHARATI PUBLIC SCHOOL, NOIDA

ACADEMIC YEAR: 2024-25

PROJECT REPORT ON

"EKLAVYA" ADVANCED AI CHATBOT

NAME	:	SAMARTH CHAUHAN
CLASS	:	XII - C
SUBJECT	:	COMPUTER SCIENCE
SUB CODE	:	083

COLLABORATORS: PRANSHU SHAVARN

DAKSH CHAUHAN

PROJECT GUIDE: MS. SOFIA GOEL

PGT (CS)

VISHWA BHARATI PUBLIC SCHOOL, NOIDA

VISHWA BHARATI PUBLIC SCHOOL, NOIDA

CERTIFICATE

This is to certify that Samarth Chauhan

Roll Number: ..... of Class XII - C has

successfully completed Computer Science Investigatory Project

according to the guidelines laid down by CBSE.

Teacher Incharge Principal

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would like to take this opportunity to extend my sincere gratitude and appreciation to my computer teacher **Ms. Sofia Goel** for providing guidance and support throughout the process of completing my computer project for school. I am also thankful to my principal **Ms. Veera Pandey** for allowing me the opportunity to explore and work on this project in the school environment.

Additionally, I would like to express my heartfelt thanks to my family for their unwavering support and encouragement during the completion of this project. The invaluable lessons and skills I have acquired through this project have enriched my learning experience and have prepared me for future challenges.

I am grateful for the opportunity to showcase my work to my class, and I am proud to have contributed to the educational environment at our school. I am truly grateful for the support and encouragement I have received throughout this project.

SAMARTH CHAUHAN

INTRODUCTION

This project involves the creation of an **AI-powered chatbot** known as **“Eklavya”** that integrates with an **SQL database** to answer users' frequently asked questions (**FAQs**). The system utilizes modern **machine learning** techniques, specifically a pre-trained **Natural Language Processing (NLP)** model, to generate **human-like responses**. The backend is developed using the **Flask web framework**, which serves as the **API** for handling user requests and responses. The chatbot leverages **SQL** to store and retrieve frequently asked questions and their corresponding answers, enabling the system to provide accurate and efficient responses for known queries.

The purpose of this project is to demonstrate how **AI** can be used to enhance user interaction with systems by integrating advanced **NLP** techniques into everyday web applications. A key feature of this project is the integration of a **SQL database**, which serves as a storage medium for **FAQs**, and the chatbot’s ability to interact with that database to pull relevant information based on the user’s query. If the question doesn’t match any pre-existing FAQ, the chatbot can generate a relevant response using **GPT-2**, a well-known **language model**.

The chatbot is designed to be **extensible**, allowing new questions and answers to be added to the database at any time. The integration of **GPT-2** ensures that the chatbot doesn’t fall back on repetitive responses but can handle a wide range of user queries, even those not explicitly listed in the database. Additionally, the use of a **web interface** allows users to interact with the chatbot in real-time, making the experience seamless and interactive.

This AI-powered chatbot can be applied across a variety of domains, including **customer service**, **educational platforms**, and even as a **personal assistant**. By combining **SQL** for database management with **machine learning models** for language understanding, this project offers a comprehensive solution for building intelligent chatbots.

PROJECT OBJECTIVES

The primary objective of this project is to create an intelligent, **AI-powered chatbot** that integrates seamlessly with an **SQL database** to answer **FAQs** and provide dynamic, context-based responses. The specific objectives of this project include:

- **Develop a Natural Language Processing (NLP) Chatbot**: The chatbot must be capable of understanding **natural language** queries posed by users and generating meaningful, **human-like responses**. It should be able to provide answers to factual questions and general conversational responses.
- **Integrate SQL Database**: A key part of this project is to implement a robust **SQL database** that stores a list of **FAQs**. This database allows the chatbot to quickly retrieve relevant answers based on user input. The SQL database must be capable of handling complex queries and provide efficient data retrieval for a fast and smooth user experience.
- **Use Pre-trained Language Models**: The chatbot will incorporate a pre-trained model, specifically **GPT-2**, to generate intelligent responses when no matching FAQ is found in the database. The ability to generate responses dynamically will ensure that the chatbot remains functional even for queries not predefined.
- **Provide an Interactive User Interface**: A clean and intuitive **web-based interface** will allow users to interact with the chatbot. The user interface will be developed using **HTML**, **CSS**, and **JavaScript**, ensuring accessibility and ease of use.
- **Ensure Scalability**: The system should be scalable, meaning that as new questions are added to the **FAQ database**, the chatbot should continue to operate seamlessly without performance issues. The chatbot should also be flexible enough to incorporate additional features such as **voice recognition** or **multi-language support** in the future.
- **Optimize Performance**: The project must be optimized for performance, ensuring that responses from the **SQL database** are returned quickly, and the **GPT-2 model** generates responses efficiently.
- **Handle User Input Effectively**: The system should be capable of processing a wide variety of user inputs, including ambiguous, complex, or vague queries, and provide relevant responses using a combination of database retrieval and text generation.

TECHNOLOGY STACK

The success of this AI-powered chatbot relies on a carefully selected stack of technologies for both the **frontend** and **backend** components. The project uses the following technologies:

1. Frontend Technologies:

- **HTML**: Used to structure the web pages that users will interact with. The **HTML** components include a user input form, a display area for the chatbot's responses, and other elements necessary for functionality.
- **CSS**: Used to style the chatbot interface, ensuring it is visually appealing and user-friendly. The styling includes layout adjustments, font choices, and colors to enhance the overall experience.
- **JavaScript**: Responsible for handling interactions between the user and the chatbot. It captures user input, sends it to the backend server, and dynamically displays the chatbot's response without needing to refresh the page.

2. Backend Technologies:

- **Flask**: A micro web framework for **Python** that simplifies the development of web applications. In this project, **Flask** is used to handle **HTTP requests**, route URLs, manage user inputs, and interact with the **SQL database**. **Flask** provides flexibility to develop **RESTful APIs**, enabling seamless communication between the frontend and backend.
- **SQL Database (SQLite or MySQL)**: The **SQL database** stores the **FAQ data**. It is queried by the backend to provide the correct answers based on user input, allowing for easy retrieval and management of FAQs. The database scales well as the chatbot's data grows.

3. Machine Learning Technologies:

- **Transformers (Hugging Face)**: The **Hugging Face Transformers** library provides pre-trained models like **GPT-2** capable of natural language understanding and generation. These models are trained on large datasets and fine-tuned to generate coherent and contextually relevant text responses.
- **Python**: The primary language used for developing both the backend and **machine learning components** of the chatbot. Python's simplicity and rich ecosystem of libraries make it ideal for this project.

4. Libraries and Frameworks:

- **Flask-SQLAlchemy**: An extension for **Flask** that simplifies integrating **SQL databases** with **Flask**. It provides an **Object Relational Mapping (ORM)** layer to simplify database operations.

SYSTEM ARCHITECTURE

The system architecture of the AI-powered chatbot is designed around a **client-server** model, where the **client (frontend)** interacts with the **server (backend)** to receive responses. This architecture ensures separation of concerns, making the system modular, scalable, and maintainable.

1. Frontend (Client-Side):

The frontend consists of an interactive interface built with **HTML**, **CSS**, and **JavaScript**. Users input their queries into a text box, and the system sends these queries to the backend server using **HTTP POST requests**. The frontend dynamically displays the chatbot's response using **JavaScript**, providing a responsive user experience. The response may either come from the **SQL database** (if the question is already stored) or from the **GPT-2 model** (if the question is new and requires generation).

2. Backend (Server-Side):

The **Flask** backend acts as the server, responsible for handling requests from the frontend and sending appropriate responses. It first attempts to match the user query with existing data in the **SQL database**. If a match is found, the corresponding answer is retrieved and sent back to the frontend. If no match is found, the **Flask backend** queries the **GPT-2 model** to generate a response, leveraging state-of-the-art **NLP techniques** to produce text that is contextually relevant to the user's query.

3. Database:

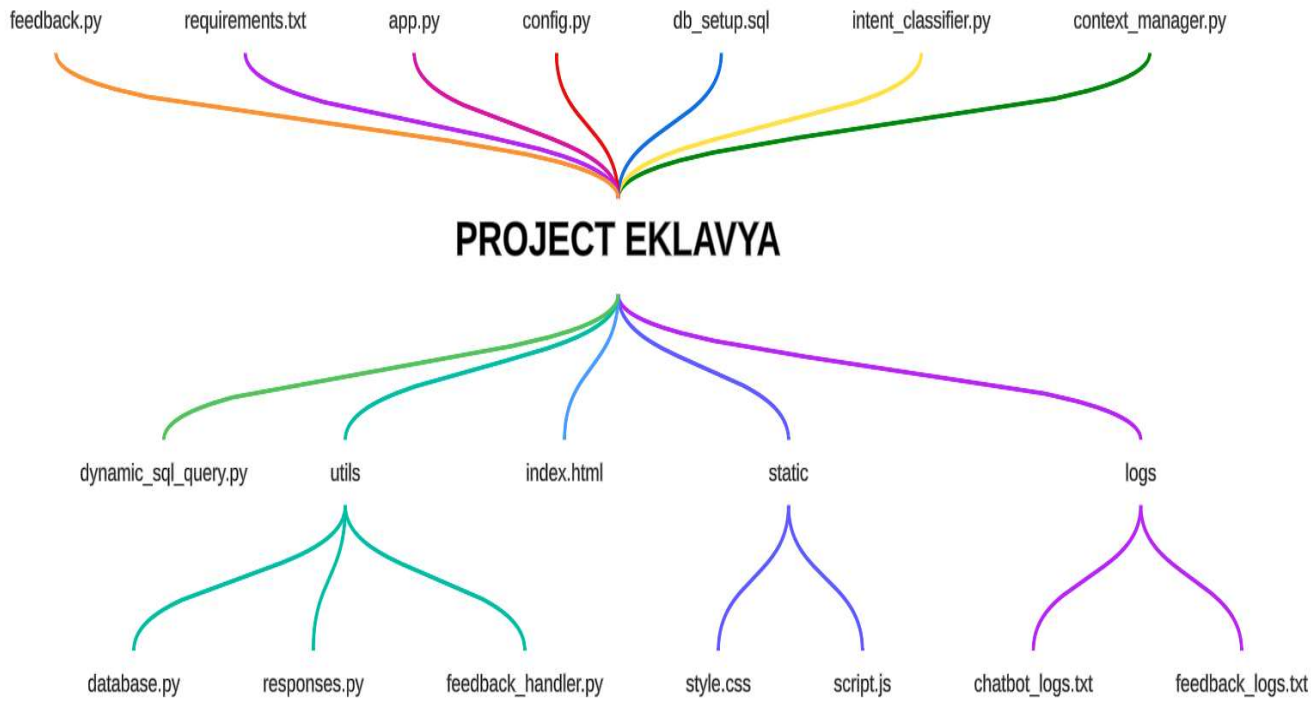
The **SQL database** stores a collection of **FAQs** that are retrieved by the **Flask backend** when a query matches. The database is designed for fast data retrieval, ensuring the chatbot responds to frequently asked questions with minimal delay.

FILE DIRECTORY SETUP

The project structure is organized to maintain clarity and modularity. Below is a recommended directory setup.

The Root Directory

The root directory contains the primary files required to execute the application, including the main scripts, configuration files, and supporting modules.



This structured layout ensures that every aspect of the project is organized and easily navigable, promoting efficient development and maintenance.

SETUP AND INSTALLATION

To set up the project, follow these steps:

Step 1: Clone the Repository

```
git clone
https://github.com/chauhan-samarth/project-eklavya.git
cd project-eklavya
```

Step 2: Create a Virtual Environment

```
python -m venv venv
venv\Scripts\activate
```

Step 3: Install Dependencies

Install the required libraries:

```
pip install -r requirements.txt
```

Step 4: Set Up Database

Set up a MySQL or SQLite database to store FAQs:

```
CREATE TABLE faq (
    id INT AUTO_INCREMENT PRIMARY KEY,
    question TEXT NOT NULL,
    answer TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Step 5: Start the Flask Application

```
python app.py
```

The application will be available at `http://localhost:5000`.

SOURCE CODE AND FILE INFORMATION

1. app.py

The `app.py` file is the central component of the chatbot application, acting as the main entry point and coordinating all interactions between the user, the backend logic, and the database. It uses Flask to create a web server, define routes for handling requests, and manage the workflow of the chatbot.

Key Functionalities:

1. **Flask Setup**:
   - Configures the Flask server to handle routes such as `/chat` and `/feedback`.
   - Enables the chatbot to process user input and manage feedback seamlessly.
2. **Intent Processing**:
   - Leverages the `IntentClassifier` class to determine the user's intent (e.g., FAQ queries, search, or general help).
   - Routes the request to the appropriate handler based on the classified intent.
3. **Context Management**:
   - Uses the `ContextManager` class to maintain user-specific conversation context.
   - Updates the context with user messages and the corresponding bot responses to ensure continuity in multi-turn conversations.
4. **Dynamic Query Execution**:
   - Integrates with the `execute_dynamic_query` function to fetch relevant data from the database dynamically.
   - Executes SQL queries to retrieve specific answers from the FAQ database or perform custom searches based on user input.
5. **Feedback Handling**:
   - Allows users to submit feedback through the `/feedback` route.
   - Uses the `record_feedback` function to log feedback into the system, including ratings and comments, for future analysis.
6. **API Gateway**:
   - Acts as a bridge between the user interface (frontend) and backend logic.
   - Processes user queries and feedback in real-time, returning appropriate responses to the frontend.

Code Walkthrough:

```
from flask import Flask, request, jsonify
from intent_classifier import IntentClassifier
from context_manager import ContextManager
from dynamic_sql_query import execute_dynamic_query
from feedback import record_feedback

app = Flask(__name__)

intent_classifier = IntentClassifier()
context_manager = ContextManager()

@app.route('/chat', methods=['POST'])
def chat():
    user_message = request.form.get('message')
    user_id = 1
    context_manager.update_context(user_id, user_message, "Bot
response here")
    intent = intent_classifier.classify_intent(user_message, ["faq",
"search", "help"])

    if intent == "faq":
        response = "This is a response from the FAQ database"
    elif intent == "search":
        response = execute_dynamic_query("SELECT * FROM faqs WHERE
question LIKE %s", (user_message,))
    else:
        response = "How can I help you?"

    return jsonify({'response': response})

@app.route('/feedback', methods=['POST'])
def feedback():
    user_id = 1
    feedback_text = request.form.get('feedback')
    rating = int(request.form.get('rating'))
    conversation_log_id = 1
```

```
    record_feedback(user_id, conversation_log_id, feedback_text,
rating)
    return jsonify({'status': 'Feedback recorded'})

if __name__ == '__main__':
    app.run(debug=True)
```

This file is the brain of the application. It determines whether the query can be answered using the database or requires AI generation. It also ensures the backend communicates seamlessly with the frontend.

2. config.py

The `config.py` file serves as a centralized configuration hub, primarily managing database connection details. This approach promotes maintainability and security by separating sensitive data and making it easy to update settings without modifying the main application code.

Key Functionalities:

1. **Database Configuration**:
    - The `DB_CONFIG` dictionary encapsulates all critical parameters needed to establish a connection to the MySQL database.
2. **Centralized Management**:
    - By placing the configuration in a single file, it simplifies updates. If the database host, credentials, or schema name changes, only this file needs editing.
3. **Separation of Concerns**:
    - Keeps sensitive details (e.g., database credentials) isolated, improving security and readability in the main application files.
4. **Scalability**:
    - The structure can easily expand to accommodate additional configuration details, such as caching mechanisms, API keys, or environment-specific settings.

Code Walkthrough:

```
DB_CONFIG = {
    'host': 'localhost',
    'user': 'root',
    'password': 'project_eklavya',
    'database': 'project_eklavya_db'
}
```

This file plays a pivotal role in ensuring the smooth operation of the application by enabling secure and efficient database connectivity.

3. Contecxt_manager.py

The `context_manager.py` file is a crucial module in the chatbot system, designed to manage conversational context for a more coherent and personalized user interaction. By storing user messages and corresponding responses, it ensures that the chatbot can handle multi-turn conversations effectively and maintain context over time.

Key Functionalities:

1. **Initialization**:
    - The `ContextManager` class initializes with a `context` dictionary to store the conversation history.
    - Each user is assigned a unique key (`user_id`) in the dictionary to maintain individual conversation threads.
2. **Context Updating**:
    - Appends the user's latest message and the corresponding chatbot response to the context dictionary.
    - Ensures continuity in conversation by storing the message-response pairs sequentially.
    - If a user's context doesn't exist, it creates a new entry for that user.
3. **Retrieve Last Message**:
    - Fetches the last message-response pair for a given user.
    - Returns `None` if the user does not exist in the context or if there is no prior interaction.
    - This is useful for referencing the previous conversation state when generating responses or maintaining flow.

Code Walkthrough:

```
class ContextManager:
    def __init__(self):
        self.context = {}

    def update_context(self, user_id, message, response):
        if user_id not in self.context:
            self.context[user_id] = []
        self.context[user_id].append((message, response))

    def get_last_message(self, user_id):
        return self.context[user_id][-1] if user_id in self.context
and self.context[user_id] else None
```

This file plays a foundational role in ensuring the chatbot feels intelligent and conversational by remembering and referencing past interactions for more engaging user experiences.

4. db_setup.py

The `db_setup.py` file serves as the backbone of the chatbot's database architecture, defining the structure and relationships of key tables that store essential data for user interactions, preferences, and feedback. This SQL script is responsible for setting up the database schema and ensuring a robust and scalable foundation for managing data efficiently.

Key Functionalities:

1. **Database Initialization**:
    - Creates a new database named `project_eklavya_db` to store all chatbot-related data.
2. **Table Creation**:
    - Defines the structure and schema for the database tables to support core features of the chatbot application.

Code Walkthrough:

```sql
CREATE DATABASE project_eklavya_db;

USE project_eklavya_db;

CREATE TABLE faqs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    question TEXT NOT NULL,
    answer TEXT NOT NULL
);

CREATE TABLE user_profiles (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    preferences TEXT
);

CREATE TABLE conversation_logs (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    message TEXT,
    bot_response TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE feedback (
    feedback_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    conversation_log_id INT,
    feedback TEXT,
    rating INT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

This script is a foundational component of the project, ensuring all necessary tables are created with appropriate relationships. By maintaining organized data storage and retrieval, it supports the application's performance, reliability, and scalability.

5. dynamic_sql_query.py

The `dynamic_sql_query.py` file is a utility module responsible for executing dynamic SQL queries in the chatbot system. By interacting with the MySQL database, it enables the chatbot to perform custom queries based on user input, allowing for flexible data retrieval that adapts to various user requests.

Key Features:

1. **Database Connectivity:**
   - Uses MySQL's `mysql.connector` to establish a connection with the database using credentials stored in `config.py`.
   - The connection parameters (host, user, password, and database) are fetched from the `DB_CONFIG` dictionary, ensuring secure and consistent access to the database.
2. **Dynamic Query Execution:**
   - The `execute_dynamic_query` function takes two parameters:
     - `query`: The SQL query string to be executed.
     - `params`: A tuple of values that will be inserted into the query, ensuring that the query is parameterized to prevent SQL injection attacks.
   - This function allows the chatbot to perform customized database queries based on user requests, such as searching the FAQ table for matching questions or retrieving user profiles based on specific criteria.
3. **Result Handling:**
   - After executing the query, the function fetches all results using `cursor.fetchall()`, which retrieves all matching rows.
   - The `cursor` is set to return results as dictionaries (i.e., column names as keys), making it easy to access the data by column name.
   - Closes both the cursor and the database connection after the query is completed to ensure proper resource management.
4. **Return Data:**
   - The results of the query (typically a list of dictionaries) are returned to the caller, allowing the chatbot to process and display the retrieved information to the user.

Code Walkthrough:

```
import mysql.connector
from config import DB_CONFIG

def execute_dynamic_query(query, params):
    connection = mysql.connector.connect(**DB_CONFIG)
```

```
    cursor = connection.cursor(dictionary=True)
    cursor.execute(query, params)
    result = cursor.fetchall()
    cursor.close()
    connection.close()
    return result
```

This file enhances the chatbot's ability to perform complex, dynamic data retrieval operations. It plays a vital role in making the system adaptable and flexible, enabling it to respond intelligently to a wide range of user requests.

6. feedback.py

The `feedback.py` file is a module designed to handle the collection and storage of user feedback within the chatbot system. This file allows users to submit their feedback regarding chatbot interactions, which can be stored in a database for future analysis and improvement of the chatbot's performance.

Key Features:

1. **Feedback Recording:**
    - The `record_feedback` function is responsible for capturing user feedback. It accepts four parameters:
        - `user_id`: The identifier of the user providing feedback.
        - `conversation_log_id`: The identifier of the conversation associated with the feedback.
        - `feedback_text`: The textual content of the feedback provided by the user.
        - `rating`: The numerical rating given by the user (typically on a scale of 1 to 5).
    - The function constructs an SQL query to insert this data into the `feedback` table of the database.
2. **Dynamic Query Execution:**
    - The feedback data is inserted into the database using the `execute_dynamic_query` function from `dynamic_sql_query.py`. This function ensures that the database interaction is dynamic and flexible, allowing the feedback to be stored securely using parameterized queries.
    - The query template `"INSERT INTO feedback (user_id, conversation_log_id, feedback, rating) VALUES (%s, %s, %s, %s)"` is used to insert the feedback data into the database. The placeholders (`%s`) are replaced by the actual values passed to the function when executed.

3. **Database Interaction:**
    - The `record_feedback` function relies on the `execute_dynamic_query` function to perform the insertion operation into the database. By using parameterized queries, it prevents SQL injection, ensuring that user input is safely handled.
4. **Feedback Submission:**
    - Once the feedback is successfully inserted into the database, it becomes available for future reporting, analysis, and chatbot improvement. This data can be used to gauge user satisfaction and identify areas where the chatbot may need refinement.

Code Walkthrough:

```
from dynamic_sql_query import execute_dynamic_query

def record_feedback(user_id, conversation_log_id, feedback_text, rating):
    query = "INSERT INTO feedback (user_id, conversation_log_id, feedback, rating) VALUES (%s, %s, %s, %s)"
    execute_dynamic_query(query, (user_id, conversation_log_id, feedback_text, rating))
```

This file plays a crucial role in collecting user input regarding the chatbot's performance. When a user completes an interaction with the chatbot, they may be prompted to provide feedback. This file processes that feedback, inserts it into the database, and ensures that it is stored for future use. This is important for improving the chatbot's accuracy and responsiveness over time, as feedback helps identify areas for improvement.

7. Index.html

The `index.html` file is the core structure of the frontend interface for the chatbot application. It provides a simple and user-friendly web page where users can interact with the chatbot, submit queries, and view the chatbot's responses. This HTML file defines the essential elements of the page, including the chat container, user input fields, and a button for

submitting messages. It also links to external CSS and JavaScript files that handle styling and interactivity.

Key Features:

1. **HTML Structure:**
    - The page is structured with a `chat-container` that houses two main components:
        - `chat-box`: A div element where the chatbot's responses will be displayed.
        - `user-input`: A text input field where the user can type their message.
        - `Send` button: A button that triggers the sending of the user's message when clicked.
2. **Responsive Layout:**
    - The `meta` tags ensure that the page is optimized for mobile devices, using a viewport setting that scales the page properly across different screen sizes.
    - The `lang="en"` attribute specifies that the content of the page is in English, ensuring proper localization and accessibility.
3. **External Resources:**
    - The `<link rel="stylesheet" href="./static/style.css">` tag links to an external CSS file that controls the styling of the chat interface, providing a polished and professional look.
    - The `<script src="./static/script.js"></script>` tag links to an external JavaScript file that defines the logic for user interactions, such as sending and receiving messages between the frontend and backend.
4. **Chatbot Interaction:**
    - The primary purpose of the `index.html` file is to facilitate real-time interaction between the user and the chatbot. Users can input text into the `user-input` field and submit their queries by clicking the "Send" button.
    - When the button is clicked, the `sendMessage()` function (defined in the `script.js` file) is called to send the user's message to the server, receive the chatbot's response, and display it in the `chat-box`.

Code Walkthrough:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

```
    <title>Project Eklavya</title>
    <link rel="stylesheet" href="./static/style.css">
</head>
<body>
    <div class="chat-container">
        <div id="chat-box"></div>
        <input type="text" id="user-input" placeholder="Ask me
anything...">
        <button onclick="sendMessage()">Send</button>
    </div>

    <script src="./static/script.js"></script>
</body>
</html>
```

This file is the front-facing component of the chatbot system, providing users with a clean, functional interface for interacting with the chatbot. It integrates the necessary HTML structure, links to external resources for styling and functionality, and enables real-time communication between the user and the backend.

8. intent_classifier.py

The `intent_classifier.py` file is a key component of the chatbot system, responsible for classifying the intent behind a user's input. By using a pre-trained zero-shot classification model from the Hugging Face library, it enables the chatbot to determine the most likely intent of a user's query without needing retraining on a specific dataset. This allows for flexible handling of different types of user queries, such as FAQs, searches, or requests for help, and ensures that the chatbot can understand a wide range of user inputs.

Key Features:

1. **Zero-Shot Classification:**
    - The `IntentClassifier` class uses the `facebook/bart-large-mnli` model for zero-shot classification, which means it can classify user queries into predefined categories without having to be retrained on specific intent data.
    - Zero-shot learning enables the model to classify new types of queries by leveraging its knowledge of general language understanding and the provided candidate labels.

2. **Model Initialization:**
    - In the `__init__` method, the classifier is initialized by loading the pre-trained BART model for zero-shot classification using Hugging Face's `pipeline`. This model is suitable for classifying text into one of several categories or "labels."
3. **Classify Intent:**
    - The `classify_intent` method accepts a user query (the text input) and a list of candidate labels (possible intents). It then runs the query through the model and returns the label (intent) that best matches the query.
    - The method uses the model's output to pick the most likely intent based on the scores, returning the highest-scoring label from the model's results.
4. **Flexible and Scalable:**
    - This classification method can be easily expanded by adding more candidate labels as the system evolves, making the chatbot capable of handling new types of intents in the future without additional retraining.

Code Walkthrough:

```
from transformers import pipeline

class IntentClassifier:
    def __init__(self):
        self.nlp = pipeline("zero-shot-classification",
model="facebook/bart-large-mnli")

    def classify_intent(self, query, candidate_labels):
        result = self.nlp(query, candidate_labels)
        return result['labels'][0]
```

This file is essential for enabling the chatbot to understand and categorize user queries into relevant intents. Using zero-shot classification, it provides a flexible and scalable way to handle a wide variety of user inputs, making it a core component of the chatbot's functionality.

9. requirements.txt

The `requirements.txt` file is a crucial component in Python projects, specifically for managing dependencies. It lists all the external libraries and their versions that are required to run the project. This file simplifies the setup process by ensuring that anyone who wants to run the project can quickly install the correct versions of all necessary libraries using `pip`. By

maintaining a list of specific versions, it ensures compatibility and prevents issues caused by version mismatches.

Key Features:

1. **Flask==2.1.0:**
   - Flask is a lightweight WSGI web application framework for Python. It is used to create the server-side components of the chatbot application.
   - **Version 2.1.0:** This specific version ensures that the project uses Flask 2.1.0, which includes important features like enhanced routing and support for async functionality. This version of Flask is chosen for its stability and compatibility with the rest of the project's dependencies.
2. **transformers==4.10.0:**
   - The `transformers` library by Hugging Face provides state-of-the-art pre-trained models for natural language processing (NLP) tasks. In this project, it is specifically used to load the BART model for zero-shot classification, which classifies user queries into predefined intents.
   - **Version 4.10.0:** This version of the library includes the necessary functionality for the task at hand, including model pipelines for text classification. By specifying this version, it ensures that the project uses a stable release of the library compatible with the other dependencies.

3. **mysql-connector-python==8.0.26:**
   - The `mysql-connector-python` library is a MySQL driver used to connect and interact with MySQL databases. In this project, it is used to manage the connection to the MySQL database, execute dynamic queries, and manage database operations such as storing and retrieving FAQs, user profiles, feedback, and conversation logs.
   - **Version 8.0.26:** This version ensures compatibility with MySQL 8.0 and provides necessary features to manage database connections efficiently.

Code Walkthrough:

```
Flask==2.1.0
transformers==4.10.0
mysql-connector-python==8.0.26
```

This file is an essential part of the project, ensuring that the necessary dependencies are installed and compatible. By locking specific versions of Flask, `transformers`, and `mysql-connector-python`, it simplifies the environment setup process and improves the stability and reproducibility of the project.

10. static/style.css

The `style.css` file defines the visual layout and appearance of the chatbot interface, ensuring a clean, user-friendly, and aesthetically pleasing design. It provides styles for various elements of the chatbot, including the chat container, message bubbles, and overall typography. By using CSS (Cascading Style Sheets), it separates the design from the content, allowing the HTML structure to focus on content and functionality while the styles dictate how the content is displayed.

Code Walkthrough:

```css
body {
  font-family: Arial, sans-serif;
  background-color: #f0f0f0;
}

.chat-container {
  width: 50%;
  margin: 50px auto;
  padding: 20px;
  background: white;
  border-radius: 5px;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
}

#chat-box {
  height: 300px;
  overflow-y: scroll;
  margin-bottom: 10px;
}

.user-msg, .bot-msg {
  padding: 10px;
  border-radius: 5px;
  margin: 5px 0;
}

.user-msg {
  background-color: #d1e7ff;
  text-align: right;
}
```

```
.bot-msg {
  background-color: #ffe8d1;
  text-align: left;
}
```

This file is an essential component of the chatbot's front-end design, ensuring that the chatbot's interface is both functional and visually appealing. By defining clear, organized styles for the chat container, messages, and overall page layout, it contributes to a smooth and intuitive user experience. The file ensures that the chatbot not only works well but also looks polished and professional.

11. static/script.js

The `script.js` file is responsible for handling the user interaction on the frontend of the chatbot interface. It enables the communication between the user's input and the backend server, ensuring that user messages are sent to the server, and the corresponding chatbot responses are displayed in the chat interface. This script primarily handles the message submission, sending requests to the backend, and updating the user interface dynamically.

Key Features:

1. **sendMessage Function:**
    - **Collect User Input:** The function retrieves the value entered by the user into the text input field (`#user-input`).
    - **Send Message to Backend:** This sends a POST request to the backend (`/chat` route) with the user's input. The message is encoded using `encodeURIComponent()` to ensure it is safely transmitted over the network.
    - **Receive Response:** Once the server processes the request, the response is returned in JSON format. This promise handles the conversion of the response into a usable JSON object.
    - **Update Chat Interface:** Upon receiving the response from the server, the function updates the chat interface. It appends the user's message and the chatbot's response into the `#chat-box` element, ensuring the messages appear in the correct sequence. After sending the message, the input field is cleared.
2. **Message Handling:**
    - **Appending Messages:**
        - The user's input is displayed on the right (`user-msg` class), and the bot's response appears on the left (`bot-msg` class). This allows users to visually differentiate between the two types of messages.

- **Scroll Handling:** Although not explicitly coded in this script, the chat container is designed to be scrollable (`#chat-box`), and the script ensures that new messages are appended in real-time.

Code Walkthrough:

```
function sendMessage() {
  const userInput =
document.getElementById('user-input').value;
  fetch('/chat', {
      method: 'POST',
      headers: { 'Content-Type':
'application/x-www-form-urlencoded' },
      body: 'message=' + encodeURIComponent(userInput)
  })
  .then(response => response.json())
  .then(data => {
      const chatBox = document.getElementById('chat-box');
      chatBox.innerHTML += `<div
class="user-msg">${userInput}</div>`;
      chatBox.innerHTML += `<div
class="bot-msg">${data.response}</div>`;
      document.getElementById('user-input').value = '';
  });
}
```

This file plays a vital role in the interactivity of the chatbot. It serves as the bridge between the frontend user interface and the backend server, managing the sending and receiving of messages. By using AJAX (`fetch`), the script ensures smooth, dynamic interactions between the user and the bot, making it an essential part of the user experience.

12. Utils/database.py

The `database.py` file is a utility module responsible for managing the connection to the MySQL database in the chatbot application. It contains a single function, `connect_db`, which

facilitates establishing a connection to the database using the configuration details provided in the `DB_CONFIG` dictionary from the `config.py` file.

Key Features:

1. **Database Connection:**
   - The `connect_db` function uses the `mysql.connector.connect()` method to establish a connection to the MySQL database.
   - The connection is made using parameters (host, user, password, and database) defined in the `DB_CONFIG` dictionary from the `config.py` file. These parameters are passed as keyword arguments to the `connect()` function, enabling dynamic configuration of the database connection.
2. **Reusability:**
   - The `connect_db` function encapsulates the logic of connecting to the database, making it reusable across other modules in the application. This ensures that any part of the application that needs to interact with the database can simply import and call `connect_db` to get a connection.
3. **Database Configuration:**
   - The connection parameters (e.g., database host, user, password, and name) are stored in a central location (`DB_CONFIG` in `config.py`). This simplifies managing and updating database credentials, as they only need to be modified in one place.

Code Walkthrough:

```
import mysql.connector
from config import DB_CONFIG

def connect_db():
    return mysql.connector.connect(**DB_CONFIG)
```

This file simplifies and centralizes the process of connecting to the database. It promotes reusability, maintainability, and efficient management of database connections within the chatbot application. By using this utility function, the application ensures that database operations can be performed smoothly and consistently throughout the system.

13. Utils/responses.py

The `responses.py` file defines a utility function, `generate_response`, which is responsible for generating appropriate responses based on the detected intent of the user's query. The function takes the intent as an input and returns a relevant message that corresponds to that intent. This file plays a crucial role in ensuring the chatbot responds in a contextually relevant and user-friendly manner.

Key Features:

1. **Intent-Based Response Generation:**
    - The `generate_response` function accepts a single parameter, `intent`, which represents the type of user query (e.g., FAQ, search, or general assistance).
    - Based on the value of the `intent`, the function returns a predefined response that corresponds to the action the chatbot is expected to perform:
        - **FAQ Intent:** If the intent is identified as "faq", the response indicates that the answer is coming from the FAQ database.
        - **Search Intent:** If the intent is identified as "search", the response informs the user that their query is being searched for.
        - **Default Response:** For any other intent, a default message is returned, offering general assistance to the user.
2. **Simplified Response Logic:**
    - The logic of generating responses based on intents is kept simple and easy to modify. This makes it easy to add more intents and their corresponding responses in the future.
    - The function can easily be extended with additional intents if the chatbot needs to handle more sophisticated scenarios.
3. **Separation of Concerns:**
    - The response generation is separated into a dedicated function, which keeps the chatbot's main flow (like intent classification or context management) clean and focused on its core tasks.
    - This separation also allows for easier maintenance and enhancement, as changes to how responses are generated are contained within this module.

Code Walkthrough:

```
def generate_response(intent):
    if intent == "faq":
        return "Here is an answer from the FAQ database."
    elif intent == "search":
        return "Searching for your query..."
    else:
```

```
return "I'm here to help you with anything!"
```

This file simplifies and centralizes the process of generating responses based on user intent. It plays an important role in enhancing the user experience by providing contextually relevant and consistent replies. The modular design of the response generation makes it easy to scale and extend as new intents and functionalities are added to the chatbot.

CONCLUSION

This project successfully develops an advanced chatbot system that combines the latest advancements in artificial intelligence, natural language processing (NLP), and database management. By leveraging Hugging Face's pre-trained BART model for zero-shot classification and integrating dynamic SQL queries, the chatbot delivers precise and contextually relevant responses, ensuring an intelligent and responsive user experience. The system’s architecture is designed to handle both structured queries from a predefined FAQ database and unstructured, complex user requests, allowing it to function as a versatile and efficient tool.

The implementation of a sophisticated context management system enables the chatbot to maintain conversation continuity, offering users a personalized interaction that adapts over multiple turns. Through careful design and modular development, the system can be easily extended and customized to accommodate new intents, integrate additional data sources, and scale as needed.

The inclusion of feedback collection and analysis further enhances the chatbot’s capabilities, allowing continuous improvement based on user interactions. This ensures the system not only meets initial functional requirements but also evolves to provide increasingly accurate and meaningful responses over time. Additionally, the project serves as a solid foundation for future advancements, such as incorporating machine learning models for deeper personalization, integrating voice interaction, or expanding the knowledge base.

In conclusion, this project demonstrates the viability and potential of combining AI-driven natural language understanding with robust backend infrastructure to create a scalable, reliable, and user-centric chatbot. It provides a high level of automation, improving operational efficiency and user engagement across various domains, such as customer service, knowledge management, and educational support. As a highly adaptable and future-proof solution, this chatbot is poised to support a wide range of applications, offering both businesses and end-users a valuable tool for interactive assistance.

REFERENCES

1. **Books & Research Papers**
   - *Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. A., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need.* Advances in Neural Information Processing Systems, 30.
     - This paper introduces the Transformer model, which is fundamental to modern NLP systems, including the GPT-based models.
   - *Chollet, F. (2017). Deep Learning with Python.* Manning Publications.
     - This book provides in-depth knowledge about deep learning techniques, including NLP and machine learning, essential for building intelligent systems like chatbots.
2. **Websites & Documentation**
   - Hugging Face Transformers. (2024). *Transformers Library*. Available at: https://huggingface.co/transformers
     - The official documentation for the Hugging Face library, which was used for integrating GPT-2 and other NLP models in the chatbot project.
   - Flask Documentation. (2024). *Flask Web Framework*. Available at: https://flask.palletsprojects.com/
     - The official Flask documentation for setting up and configuring the web framework used in the chatbot server.
   - MySQL Documentation. (2024). *MySQL Database*. Available at: https://dev.mysql.com/doc/
     - The official MySQL documentation for understanding database interactions and integration with Python via the `mysql-connector-python` library.
3. **Online Courses & Tutorials**
   - *Python for Data Science and Machine Learning Bootcamp* by Jose Portilla, available on Udemy.
     - A comprehensive course on Python programming, data science, and machine learning, which helped in understanding AI techniques and integrating them into the chatbot.
   - *Building Chatbots with Python and Flask* by Real Python. Available at: https://realpython.com/build-a-chatbot-python-flask/
     - This tutorial provided guidance on how to use Python and Flask to build chatbots, which is foundational for the implementation of the chatbot system in this project.
4. **Tools & Libraries Used**
   - *Transformers* by Hugging Face. Available at: https://huggingface.co/
     - The Transformers library was used to implement the AI models, including GPT-2 for generating responses.
   - *Flask* Web Framework. Available at: https://palletsprojects.com/p/flask/

        - Flask is the Python web framework used for building the backend of the chatbot system.
    - *MySQL Connector for Python*. Available at: https://pypi.org/project/mysql-connector-python/
        - MySQL Connector is used for connecting Python to MySQL databases, enabling the chatbot to interact with a database for FAQs and logging conversations.

5. **Miscellaneous**
    - *Stack Overflow*. Available at: https://stackoverflow.com/
        - A great resource for troubleshooting and finding solutions to various programming challenges encountered during the project.