

A PROJECT REPORT ON

“AI-CHAT(CHATGPT)”

*Submitted in partial fulfilment of
Master in
Computer Application*



Session: 2021-2023

DEPARTMENT OF COMPUTER APPLICATION

Makhanlal Chaturvedi National University of Journalism and Communication
Bhopal

SUBMITTED TO:

Dr. MANISH MAHESHWARI
(Head of Computer Department)

SUBMITTED BY:

BHARTI JHA
M.C.A 4th SEM
ENROLLMENT NO.:
AX1020999020

DECLARATION

I hereby declare that the mini project work being presented in this report entitled **“AI-CHAT(chatgpt)”** submitted in the department of Computer Science, **FACULTY OF TECHNOLOGY** Makhanlal Chaturvedi National University of Journalism and Communication, Bhopal is the authentic work carried out by me under the guidance of **Dr. Manish Maheshwari**, professor of Computer Science, Makhanlal Chaturvedi National University of Journalism and Communication, Bhopal.

Date: 29 MAY, 2023

BHARTI JHA
MCA 4th SEM
Department of Computer Science
MCNUJC, Bhopal-462044(MP)

CERTIFICATE

This is to certify that this report represents the original work done by “**BHARTI JHA**” during this project submission as a partial fulfilment of the requirement for the **AI CHAT(CHATGPT)** Project of Masters of Computer Application, IV Semester, of the **Makhanlal Chaturvedi National University of Journalism and Communication, Bhopal (MP-462011)**.

Prof.Dr.Manish Maheshwari
Head of Computer Department
MCNUJC, Bhopal-462044(MP)

External Examiner

Date:29 may 2023
Place:-Bhopal

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my teacher **Dr. Manish Maheshwari** who gave me the golden opportunity to do this wonderful project on the topic “**AI-CHAT(CHATGPT)**”, which also helped me in doing a lot of research and I came to know about so many new things I am really thankful of them.

Secondly I would also like to thank my professors Nitin sir and Ravi Mohan sir. Who helped me a lot in finalizing this project within the limited time frame.

Date: 26 may 2023

Bharti Jha
MCA 4th SEM.
Department of Computer Science
MCNUJC, Bhopal-462044 (MP)

TABLE OF CONTENTS

TITLE	PAGE NO.
DECLARATION	2
CERTIFICATE	3
ACKNOWLEDGEMENT	4
 CHAPTER 1 INTRDUCTION	 6
1.1 ABSTRACT	6
1.2 OBJECTIVES	7
1.3 PURPOSE	8
1.4 MAIN FEATURES	9
1.5 TARGET AUDIENCE	10
1.6 HARDWARE AND SOFTWARE REQUIREMENT	11
 CHAPTER 2 (LIBRARY AND FRAMEWORK)	 12
 CHAPTER3 (FLOW CHART)	 13
 CHAPTER 4. PROJECT STRUCTURE	 15
4.1 MODELS	17
4.2 VIEWS	19
4.3 TEMPLATES	25
4.4 STATICFILES	54
 CHAPTER 5. DATABASE DESIGN	 56
 CHAPTER 6. USER AUTHENTICATION AND AUTHORIZATION	 60
 CHAPTER 7. KEY FEATURES AND FUNCTIONALITY	 62
 CHAPTER 8. APIS AND WEB SERVICES	 64
 CHAPTER 9. TESTING AND QUALITY ASSURANCE	 66
 CHAPTER 10. DEVELOPMENT ENVIRONMENT AND TECHNOLOGIES	 69
 CHAPTER 11. FUTURE PLANS AND ENHANCEMENTS	 71
 CHAPTER 12. FALIURE	 73

CHAPTER 1

INTRODUCTION

GPT stands for (Generate Pre-Trained model) The CHATGPT project is a Django-based web application that utilizes the capabilities of the ChatGPT language model developed by OpenAI. Its purpose is to provide users with an interactive chat interface where they can have natural language conversations with the AI model.

1.1 ABSTRACT

The abstract of the CHATGPT project is as follows:

The CHATGPT project is a Django-based web application that enables users to engage in interactive and natural language conversations with the ChatGPT language model. Through a user-friendly chat interface, users can input text messages and receive real-time responses from the AI model. The project utilizes advanced natural language processing capabilities to understand user inputs and generate contextually relevant and coherent responses. With customizable prompts and multi-domain knowledge, users can explore various topics and enjoy a chatbot-like experience. The CHATGPT project serves as a platform for developers, researchers, and individuals interested in conversational AI to interact with the ChatGPT model and showcase the potential of natural language understanding and generation.

1.2 OBJECTIVES

The objectives of the CHATGPT project are as follows:

Develop a User-Friendly Interface: Create an intuitive and user-friendly chat interface that allows users to easily input text messages and receive responses from the ChatGPT model. Focus on providing a seamless and engaging user experience.

Implement Natural Language Processing: Utilize the advanced natural language processing capabilities of the ChatGPT model to understand user inputs, handle context, and generate coherent and contextually relevant responses. Strive for a high level of conversational fluency.

Enable Customization and Personalization: Allow users to customize prompts or messages to guide the conversation with the AI model. Implement features that enable personalization and tailoring of the model's responses to specific needs or topics.

Ensure Context Management: Develop mechanisms to maintain context throughout the conversation, enabling the model to provide context-aware responses. Implement techniques to handle multi-turn interactions effectively.

Showcase Multi-Domain Knowledge: Demonstrate the versatility of the ChatGPT model by enabling conversations across various domains, including general knowledge, entertainment, technology, and more. Ensure the model can provide informative and accurate responses across a wide range of topics.

Gather User Engagement Insights: Implement metrics and analytics to gather insights on user engagement, satisfaction, and common queries. Use this data to continually improve the application and enhance user experiences.

Provide a Platform for Exploration: Create a platform where developers, researchers, and individuals interested in conversational AI can interact with the ChatGPT model, customize prompts, and experiment with different conversational scenarios. Foster a community for sharing insights and advancements in the field.

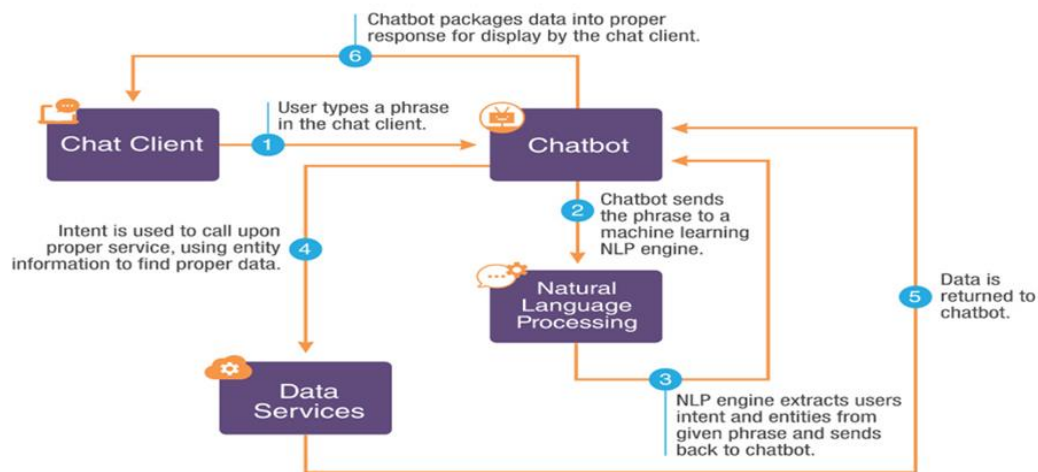
Ensure Scalability and Performance: Design the application to handle a high volume of concurrent users and deliver responses in a timely manner. Optimize performance to provide a smooth and responsive chat experience.

1.3 PURPOSE

The purpose of the CHATGPT project is to provide users with an interactive and conversational experience using the ChatGPT language model. By utilizing natural language processing and advanced AI capabilities, the project aims to enable users to have meaningful and engaging conversations with the AI model in real-time.

The project's purpose extends beyond simple question-and-answer interactions by focusing on creating a chatbot-like experience where users can engage in fluid conversations, receive contextually relevant responses, and explore the model's knowledge across various domains. It aims to showcase the potential of natural language understanding and generation in creating interactive and intelligent applications.

Overall, the purpose of the CHATGPT project is to provide a user-friendly and interactive interface that leverages the power of the ChatGPT model, enabling users to engage in natural language conversations and experience the potential of AI-driven communication.



(Fig: AI-CHAT(chatgpt) processing)

1.4 MAIN FEATURES

The main features of the CHATGPT project include:

Interactive Chat Interface: The project provides a user-friendly chat interface where users can enter text messages and receive real-time responses from the ChatGPT model. The interface aims to simulate a natural conversation experience.

Natural Language Processing: The project leverages the advanced natural language processing capabilities of the ChatGPT model to understand and interpret user inputs. It utilizes techniques such as language modeling, contextual understanding, and semantic analysis to generate coherent and contextually relevant responses.

Context Management: The application maintains context across multiple user interactions, allowing for more meaningful and coherent conversations. It considers previous user inputs and model responses to provide context-aware replies, creating a more interactive and engaging experience.

Customizable Prompts: Users have the ability to customize prompts or initial messages to guide the conversation with the AI model. This feature allows users to set the tone, topic, or specific instructions for the AI's responses, tailoring the conversation to their preferences or needs.

Multi-domain Knowledge: The ChatGPT model has been trained on a wide range of topics, enabling it to engage in conversations spanning various domains. The project takes advantage of this capability, allowing users to explore different topics and receive informative and accurate responses.

Real-time Response: The project strives to deliver responses from the ChatGPT model in real-time, providing users with immediate feedback and maintaining the conversational flow.

Error Handling and Fallback Mechanisms: The application implements error handling and fallback mechanisms to handle situations where the model might not understand or generate appropriate responses. It aims to gracefully handle such scenarios and provide helpful messages to users.

These main features collectively create an interactive and engaging chat experience, leveraging the capabilities of the ChatGPT model to deliver contextually relevant and coherent responses to user inputs.

1.5 TARGET AUDIENCE

The target audience for the CHATGPT project can include:

Developers and AI Enthusiasts: The project caters to developers and AI enthusiasts who are interested in exploring the capabilities of the ChatGPT model and conversational AI. They can use the project as a learning resource, experiment with customization options, and gain insights into natural language understanding and generation.

Researchers: The project can be valuable for researchers in the field of natural language processing and conversational AI. It provides a platform to study user interactions, evaluate the performance of the ChatGPT model, and contribute to advancements in the field.

Chatbot Designers: Chatbot designers and conversational UI/UX experts can benefit from the project by understanding the capabilities and limitations of the ChatGPT model. They can gather insights on conversational flow, context management, and user engagement, which can be applied to design more effective and interactive chatbot experiences.

Technology Enthusiasts: Individuals with a general interest in technology and AI may find the project intriguing and engaging. They can interact with the ChatGPT model, explore its conversational abilities, and experience the potential of AI-driven conversations firsthand.

Education and Training: The project can serve as an educational tool for teaching natural language processing, AI, and chatbot development. Students and educators can utilize the project to learn about conversational AI concepts, experiment with customization, and understand the challenges and opportunities in building interactive chat systems.








General Users: While the project targets a technical audience, general users who are curious about conversational AI may also find it interesting. They can engage in conversations with the ChatGPT model, seek information, or simply enjoy the interactive and simulated chat experience.

The CHATGPT project aims to cater to a diverse audience interested in conversational AI, including developers, researchers, chatbot designers, technology enthusiasts, educators, and general users. It provides an accessible platform to interact with the ChatGPT model and explore the potential of AI-driven conversations.

1.6 Hardware requirement

1. Processor:	A multicore processor with a clock speed of 833 GHz or higher is recommended.
2. RAM:	At least 4 GB of RAM is recommended, although more RAM will be beneficial for handling larger models and concurrent requests.
3. Storage:	10GB

Software requirement

-  Python: 3.10.4 version
-  Pip:24.1.2 version
-  Django: 4.2 version
-  ChatGPT Dependencies: OpenAI
-  OpenAI:0.27.4 version
-  Text-Editor:Visual Studio
-  Model:text-davinci-003

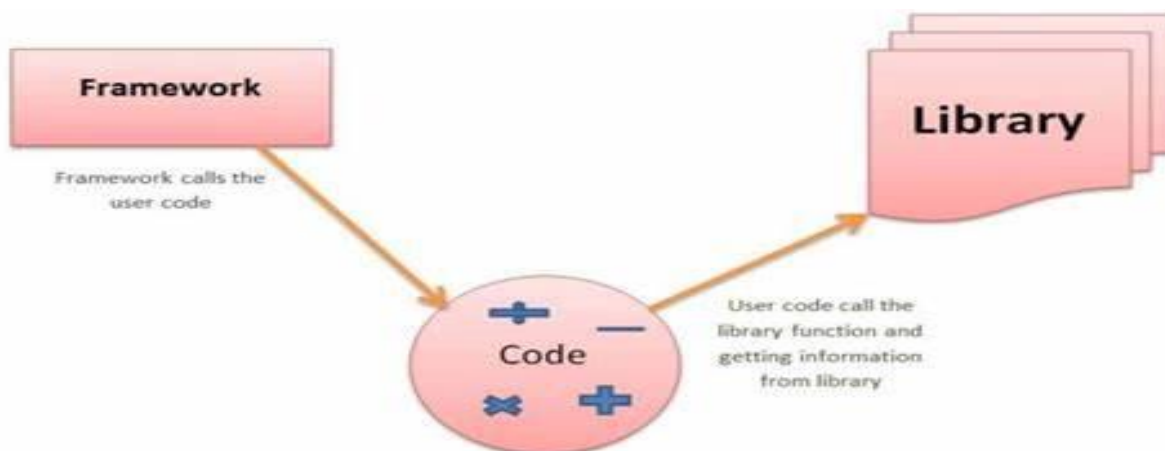
1. OPERATING SYSTEM	WINDOWS 10
2. PLATEFORM	VISUAL STUDIO
3. LANGUAGE	PYTHON
4. FRONT-END	HTML,CSS,JAVASCRIPT,BOOTSTRAP
5. BACK-END	PYTHON
6. FRAMEWORK	DJANGO

CHAPTER 2

Library and framework

The libraries and frameworks used in the code are as follows:

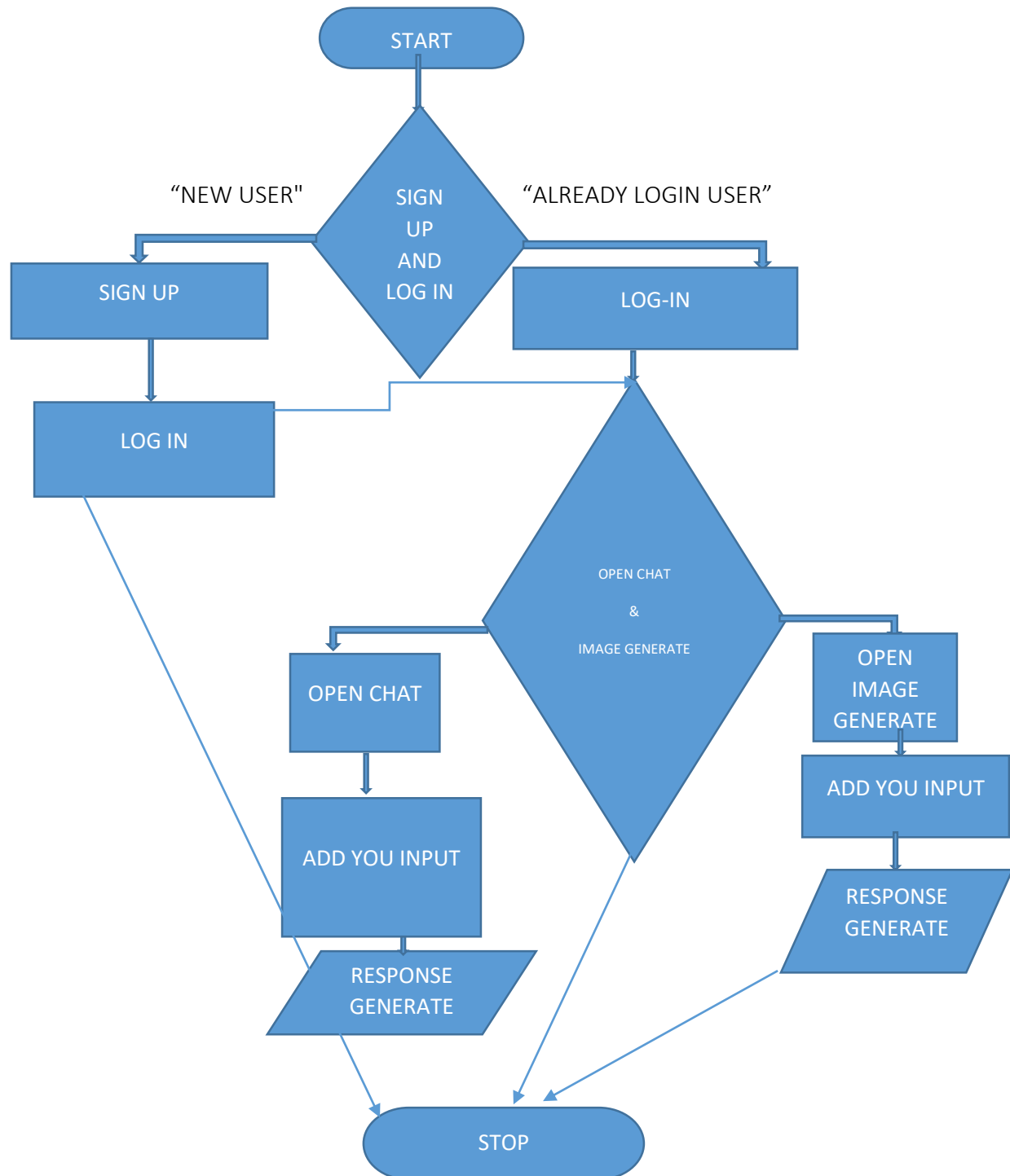
1. **Django:** The web framework used for building the application.
2. **OpenAI:** The OpenAI API library used for making requests to the ChatGPT model.
3. **Dotenv:** A library for loading environment variables from a .env file.
4. **Requests:** A library for making HTTP requests to retrieve image data.
5. **re:** A library for working with regular expressions.
6. **Django's built-in libraries:** These include various modules from **Django**, such as `render`, `redirect`, `authenticate`, `login`, `logout`, `messages`, `UserCreationForm`, `ContentFile`, and the `models` module for database interactions.
7. **PyTorch:** If you are using PyTorch as the deep learning framework for your ChatGPT model, you will need to import the necessary PyTorch modules for building, training, and interacting with the model.
8. **Transformers:** The Transformers library is a popular open-source library for natural language processing that provides a range of pre-trained language models like **GPT-2** and **GPT-3**. These pre-trained models can be fine-tuned for specific tasks like generating responses in a chatbot system.
9. **Hugging Face:** Hugging Face is a startup that has created several popular open source libraries for natural language processing like Transformers and Tokenizers. The Hugging Face libraries provide easy-to-use APIs for working with pre-trained language models and other NLP tasks. The Tokenizers library is an open-source library for tokenizing text data. Tokenization is the process of splitting text data into smaller units called tokens, which are then used as inputs to language models like GPT.



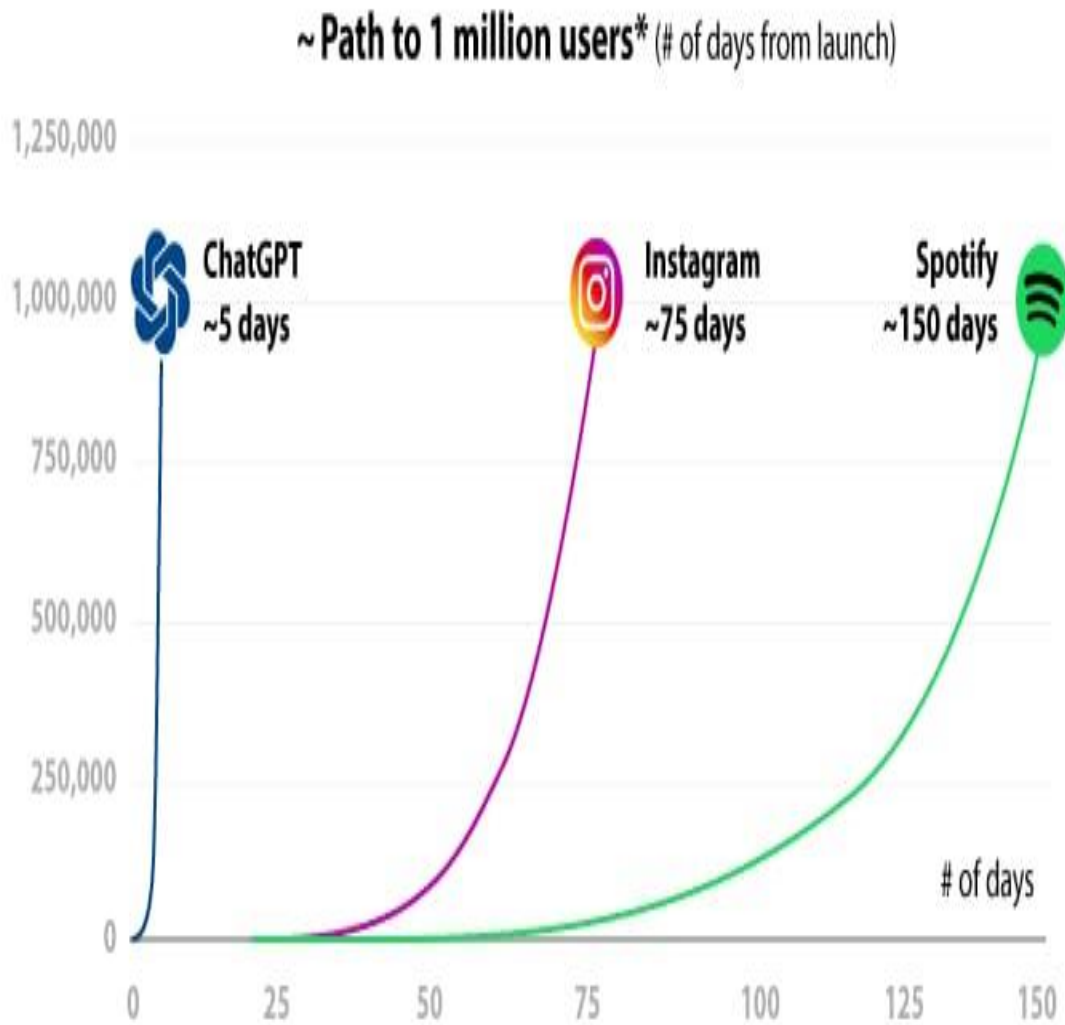
(fig: Library and framew

CHAPTER 3

FLOW CHART



(Fig: show flow chart how to working project)



(Fig: How much is growing)

CHAPTER 4

PROJECT STRUCTURE

The structure of a Django project typically follows a standard convention to maintain organization and modularity. Here is a general outline of the project structure:

Project Root:

- This is the top-level directory that contains the entire Django project.
- It usually has the same name as your Django project.
- It may contain files like **manage.py**, **.env** (for environment variables), and project **configuration files**.

Application(s):

- Django projects are composed of one or more applications.
- Each application serves a specific functionality or module within the project.
- Applications are typically organized as separate directories within the project root.
- Each application may include files like **models**, **views**, **templates**, **static files**, and **migrations**.

Project Configuration:

- The project root contains files related to project-wide configuration.
- **settings.py**: Contains project settings, including database configurations, installed applications, middleware, static files, and more.
- **urls.py**: Defines URL patterns and routes requests to the appropriate views within the project.

Static Files:

- The static files directory holds CSS, JavaScript, images, and other static assets used in the project.
- It is typically located within each application directory or in a central location in the project root.
- Static files can be organized into subdirectories based on their purpose (e.g., CSS, JS, images).
- **Templates:**
 - The templates directory contains HTML templates used for rendering dynamic content.
 - It can be located within each application directory or in a central location in the project root.
 - Templates can be organized into subdirectories based on the application or functionality they serve.

Database:

Django supports different database backends, and the configuration is specified in the **settings.py** file.

Database migrations, created using Django's **migration** system, are stored in the respective application directories.

Additional Files:

Other files that exist in the project structure include:

requirements.txt: Lists the required packages and dependencies for the project.

README.md: Provides information and documentation about the project.

Procfile (for deployment): Specifies commands for running the application in a production environment.

It's important to note that the exact structure of your Django project may vary depending on your specific requirements, organization preferences, and additional libraries or frameworks you choose to use.

When creating a Django project, you can use the **django-admin startproject** command to set up the initial project structure automatically, and then create applications within the project using the **python manage.py startapp** command.

4.1 MODELS

In a Django project, models define the structure and behavior of the data that will be stored in the database. They represent the entities or objects in your application and define the fields and relationships associated with them. Here's an explanation of the models in Django:

Definition:

Models are typically defined in a `models.py` file within each application directory. They are Python classes that inherit from the `django.db.models.Model` base class. Each model class represents a database table, and each attribute represents a table column.

Fields:

Fields define the data types and characteristics of the model's attributes (columns). Django provides a wide range of built-in field types, including integers, strings, dates, booleans, etc. You can also define relationships between models using fields like Foreign Key, `OneToOneField`, or `ManyToManyField`.

Database Sync:

After defining models, you need to create corresponding database tables. Django provides a database migration system to handle schema changes and keep the database in sync with the models. Migrations are created using the `python manage.py makemigrations` command and applied using `python manage.py migrate`.

Model Queries:

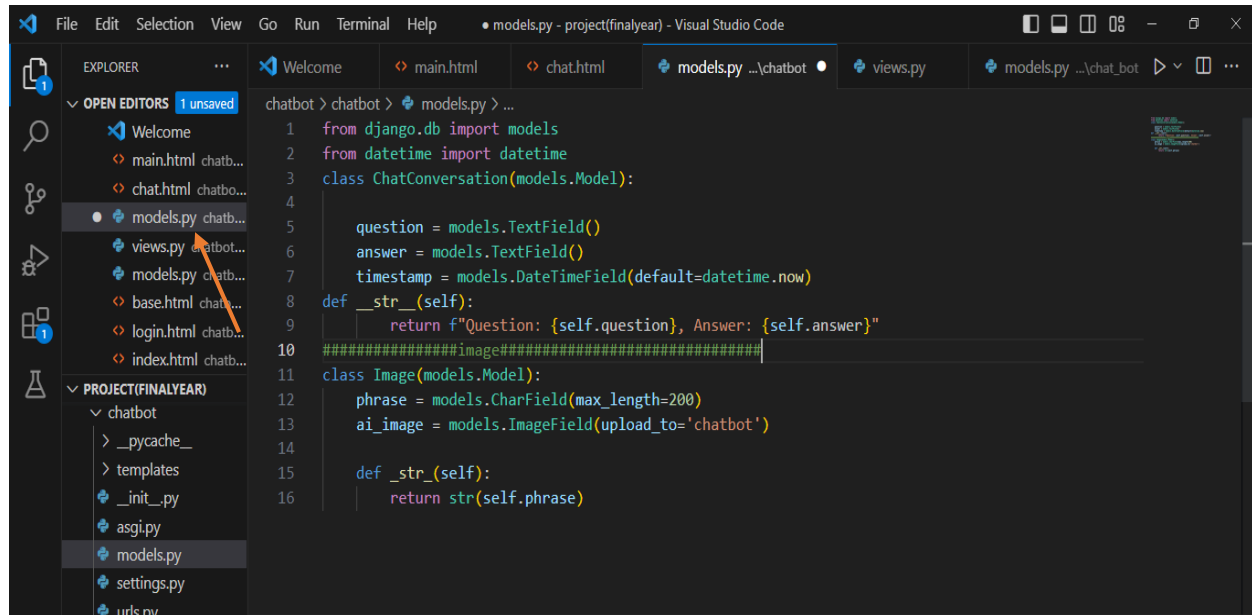
Models enable you to perform database queries and retrieve, create, update, or delete records. Django provides a powerful ORM (Object-Relational Mapping) that abstracts the database operations. You can use methods and querysets on model classes to perform complex database queries, filtering, ordering, and aggregations.

Admin Interface:

Django's **admin** interface allows easy management of the models and their data. By registering models in the `admin.py` file, you get an automatically generated **CRUD** (Create, Read, Update, and Delete) interface. This interface provides a user-friendly way to interact with the data stored in the database.

Models are a fundamental part of a Django project as they define the structure and relationships of the data. By utilizing models, easily interact with the database and perform various operations on the data.

Example:



Models.py

```
from django.db import models
from datetime import datetime
class ChatConversation(models.Model):

    question = models.TextField()
    answer = models.TextField()
    timestamp = models.DateTimeField(default=datetime.now)
def __str__(self):
    return f'Question: {self.question}, Answer: {self.answer}'
#####image#####
class Image(models.Model):
    phrase = models.CharField(max_length=200)
    ai_image = models.ImageField(upload_to='chatbot')

    def __str__(self):
        return str(self.phrase)
```

4.2 VIEWS

Views handle the logic for processing user requests and generating responses. They receive HTTP requests, interact with models and other components, and return HTTP responses.

Here's an explanation of views in Django:

Definition:

- Views are Python functions or class-based views that define how a web application responds to a particular URL pattern or request.
- They are typically defined in a **views.py** file within each application directory.

1. Function-based Views:

- Function-based views are defined as Python functions that accept an **HTTP request** as a parameter and return an **HTTP response**.
- Inside the view function, you can access and manipulate data from models, interact with external APIs, perform calculations, etc.
- You can use Django's shortcuts, such as **render()** for rendering templates or **redirect()** for URL redirection.

2. Class-based Views:

- Class-based views are an alternative approach where views are defined as classes instead of functions.
- Django provides various generic class-based views that implement common functionalities, such as creating, updating, or deleting objects.
- Class-based views offer code reusability, mixins for extending functionality, and easy handling of common HTTP methods.

3. URL Mapping:

- Views are mapped to specific **URLs** using Django's URL configuration.
- URLs are defined in the **urls.py** file within each application directory or in the project's main **urls.py** file.
- URL patterns are associated with view functions or classes, allowing Django to route incoming requests to the appropriate view.

4. Context and Templates:

- Views can pass data (context) to templates for rendering dynamic content in the response.
- Context is typically a dictionary containing variables that can be accessed in the template.
- Views use the **render()** function to combine the template with the context and return an HTTP response.

VIEWS.PY

```

chatbot > chat_bot > views.py > chatgpt
1  from django.shortcuts import render, redirect
2  from django.contrib.auth import authenticate, login, logout
3  from django.contrib import messages
4  from django.contrib.auth.forms import UserCreationForm
5  import openai,os,requests
6  from dotenv import load_dotenv
7  from django.core.files.base import ContentFile
8  from chat_bot.models import Image
9  import re
10 from .models import ChatConversation
11
12 def chatgpt(request):
13     chatgpt_response = ""
14     regenerate = False
15     chatgpt_response_lines = []
16
17     if request.method == 'POST':
18         user_input = request.POST.get('user_input')
19         context = request.POST.get('context', "")
20         regenerate = bool(request.POST.get('regenerate', False))

```

```

chatbot > chat_bot > urls.py > ...
1
2  from django.urls import path
3  from . import views
4  from django.conf import settings
5  from django.conf.urls.static import static
6
7
8  urlpatterns = [
9
10     path('', views.index_view, name='index_view'),
11     path('login/', views.login_view, name='login_view'),
12     path('signup/', views.signup_view, name='signup_view'),
13
14     path('logout/', views.logout_view, name='logout'),
15     path('home/', views.home_view, name='home'),
16     path('generate_image_from_txt/', views.generate_image_from_txt, name="generate_image_from_txt"),
17     path('chatgpt/', views.chatgpt, name='chatgpt'),
18     path('chat/clear/', views.chatgpt, name='clear_chat'),
19

```

VIEWS.PY

```

from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib import messages
from django.contrib.auth.forms import UserCreationForm
import openai,os,requests
from dotenv import load_dotenv

```

```

from django.core.files.base import ContentFile
from chat_bot.models import Image
import re
from .models import ChatConversation

def chatgpt(request):
    chatgpt_response = ""
    regenerate = False
    chatgpt_response_lines = []

    if request.method == 'POST':
        user_input = request.POST.get('user_input')
        context = request.POST.get('context', "")
        regenerate = bool(request.POST.get('regenerate', False))

        if regenerate:
            prompt = f"{context}\nUser: {user_input}\nAI:"
        else:
            prompt = f"{context}\nUser: {user_input}\nAI: {chatgpt_response}"

        response = openai.Completion.create(
            engine='text-davinci-003',
            prompt=prompt,
            max_tokens=1024,
            temperature=0.5,
            n=1,
            stop=None,
            timeout=1,
        )

        chatgpt_response = response.choices[0].text.strip()
        chatgpt_response = chatgpt_response.replace('\n', '<br>') # Replaced \n with HTML line
        break
        chatgpt_response_lines = re.split('<br>', chatgpt_response)

        question = f"User: {user_input}"
        answer = f"AI: {chatgpt_response}"

        if user_input:
            ChatConversation.objects.create(question=question, answer=answer)

```

```
conversation_history = ChatConversation.objects.order_by('timestamp')
```

```
if request.GET.get('clear'):
    ChatConversation.objects.all().delete()
    return redirect('chatgpt')
```

```
return render(request, 'chat.html', {
    'chatgpt_response': chatgpt_response,
    'chatgpt_response_lines': chatgpt_response_lines,
    'regenerate': regenerate,
    'conversation_history': conversation_history
})
```

```
#####Log-in #####
```

```
def login_view(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
```

```
    return redirect('home')
```

```
else:
    messages.error(request, 'Invalid username or password.')
    return render(request, 'login.html')
```

```
def logout_view(request):
    logout(request)
    return redirect('login')
```

```
def signup_view(request):
    form = UserCreationForm()
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Account created successfully.')
```

```
    return redirect('login')
```

```

else:
    messages.error(request, 'There was an error creating your account.')
    return render(request, 'signup.html', {'form': form})

def home_view(request):
    return render(request, 'home.html')
def index_view(request):
    return render(request, 'index.html')
#images
# Load environment variables from .env file
load_dotenv()
api_key = os.getenv("OPENAI_KEY")
openai.api_key = api_key
def generate_image_from_txt(request):
    obj = None
    if api_key is not None and request.method=="POST":
        user_input = request.POST.get('user_input')

    if user_input and isinstance(user_input, str):
        response = openai.Image.create(

        prompt=user_input,
        size='256x256',

        )

    img_url = response["data"][0]["url"]
    #print(img_url)

    response = requests.get(img_url)
    img_file = ContentFile(response.content)
    count = Image.objects.count() + 1
    fname = f"image-{count}.jpg"
    obj = Image(phrase=user_input)
    obj.ai_image.save(fname, img_file)
    obj.save()
    print(obj)
    return render(request, "main.html", {"object": obj})

```

URLS.PY

```
from django.urls import path
from . import views
from django.conf import settings
from django.conf.urls.static import static
urlpatterns = [
    path('', views.index_view, name='index_view'),
    path('login/', views.login_view, name='login_view'),
    path('signup/', views.signup_view, name='signup_view'),
    path('logout/', views.logout_view, name='logout'),
    path('home/', views.home_view, name='home'),
    path('generate_image_from_txt/', views.generate_image_from_txt, name="generate_
image_from_txt"),
    path('chatgpt/', views.chatgpt, name='chatgpt'),
    path('chat/clear/', views.chatgpt, name='clear_chat'),
    ]+ static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

URLS.PY

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
urlpatterns = [
    path('admin/', admin.site.urls),
    path("", include('chat_bot.urls')),
]
```


4.3 TEMPLATES

Templates are used to generate dynamic HTML pages that are rendered and sent as responses to user requests. Templates allow you to separate the presentation logic from the business logic of your application.

Here's an explanation of templates in Django:

Definition:

- **Templates** are **HTML files** with embedded **Django template language** (DTL) syntax.
- They are typically stored in a templates directory within each application directory or in a central location specified in the **project's settings**.

a) **Rendering:**

- Templates are **rendered** by views to generate dynamic content.
- Views pass data (context) to templates, which can be used to populate placeholders, conditionally display content, or iterate over lists of data.
- Django's template engine processes the template, evaluates the template tags and filters, and generates the final **HTML output**.

b) **Template Language:**

- Django's **template language** (DTL) is a simple and expressive language used within templates.
- DTL allows you to perform logic operations, iterate over lists, access variables, apply filters to data, and more.
- Template tags (enclosed in `{% %}`) and filters (enclosed in `{{ }}`) are used to perform these operations.

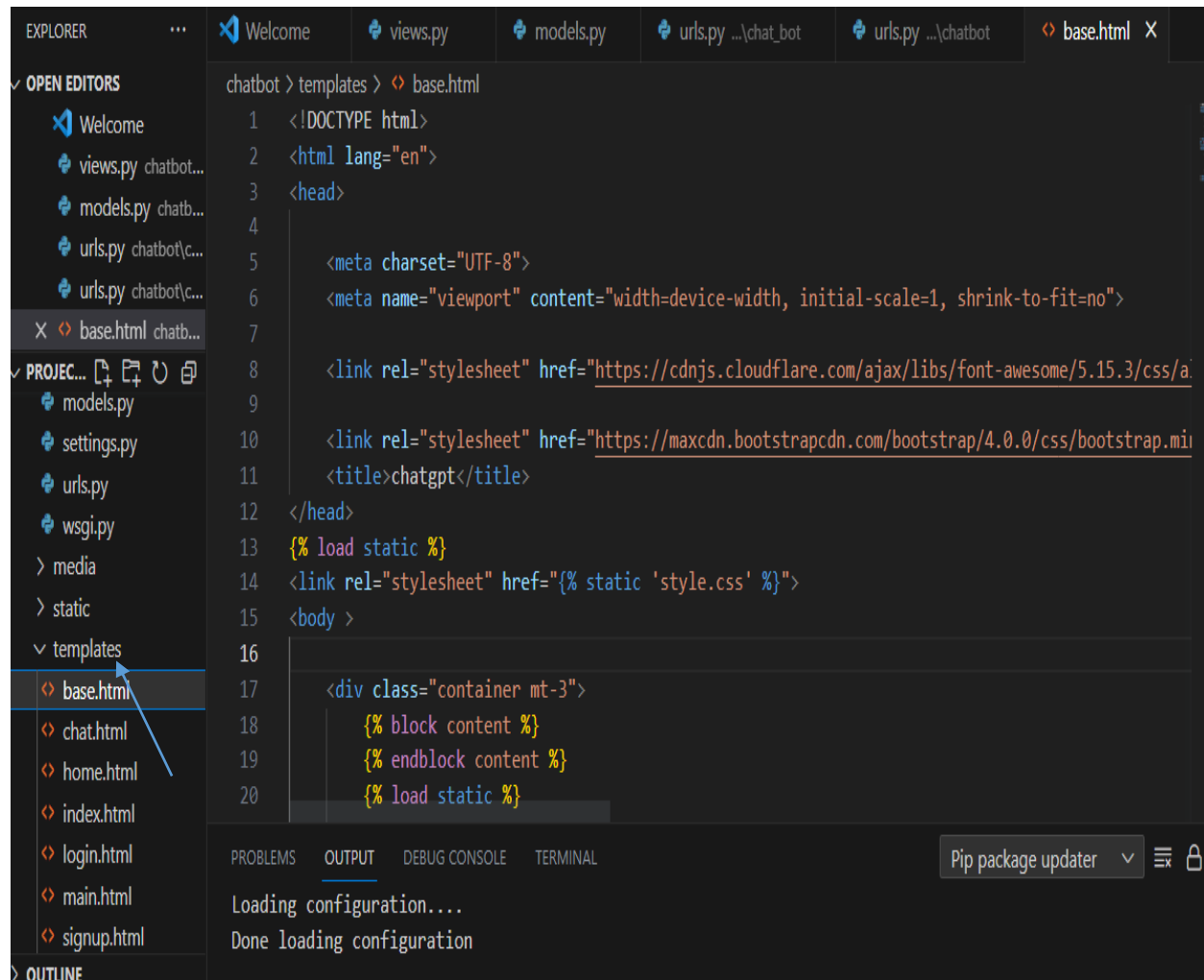
c) **Template Inheritance:**

- Templates can be structured using inheritance to promote code reuse and maintain consistency.
- **Base templates** define the overall structure and common elements of a website.
- Child templates can extend or override specific sections of the base template to provide customized content.

d) **Static Files and Media:**

- Templates can reference **static files** (CSS, JavaScript) and **media files** (images, videos) using **static file** handling in Django.
- Static file URLs can be generated using the `{% static %}` template tag, which resolves to the appropriate URL based on the static file configuration.

- Media files, such as user-uploaded images, can be served using the **MEDIA_URL** setting and appropriate configurations.



(fig:TEMPLATES)

Base.html

```

<!DOCTYPE html>
<html lang="en">
<head>

    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-
to-fit=no">

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.15.3/css/all.min.css" integrity="sha512-
GoPRAkgdHQLZQ1JTWj194aK67/ZsCQLJb/WyP8FjwGkHejKd80TksTfT+7GpDmht+AW7VJtdpx6TZf61m
klKYQ==" crossorigin="anonymous" referrerpolicy="no-referrer" />
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
    <title>chatgpt</title>
</head>
{% load static %}
<link rel="stylesheet" href="{% static 'style.css' %}">
<body >

    <div class="container mt-3">
        {% block content %}
        {% endblock content %}
        {% load static %}
    </div>

</body>
</html>

```

Index.html

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Welcome to ChatGpt</title>
    {% load static %}
    <script src="https://kit.fontawesome.com/f3927a3561.js"
crossorigin="anonymous"></script>

    <link rel="stylesheet" href="{% static 'style.css' %}">
</head>
<body>
    <center>

```

```


<h1>Welcome to ChatGpt</h1>
<p>Log in with your OpenAI account to continue</p>

<div class="container">
  <form method="post" action="{% url 'login_view' %}">
    {% csrf_token %}
    <button type="submit" class="button">
      <i class="fa-solid fa-right-to-bracket"></i>Login</button>
    </form>
    <form method="post" action="{% url 'signup_view' %}">
      {% csrf_token %}
      <button type="submit">
        <i class="fa-solid fa-user-plus"></i>Signup</button>
      </form>
    </div>
<h2>Don't have an OpenAI account? <a href="https://beta.openai.com/signup/">Sign
up here</a></h2>
</center>
</body>
</html>
<style>
  p {
    color: #fff;
    text-shadow:
      0 0 5px #fff,
      0 0 10px #fff,
      0 0 20px #fff,
      0 0 40px #0ff,
      0 0 80px #0ff,
      0 0 90px #0ff,
      0 0 100px #0ff,
      0 0 150px #0ff;
  }

  :root {
    --primary-color: #4CAF50;
    --background-color: rgba(30, 8, 52, 0.511);
  }

  body {
    background-color: var(--background-color);
    padding: 20px;
  }

```

```
form {
  display: flex;
  flex-direction: column;
  align-items: center;
  margin-top: 50px;
}

button {
  background-color: var(--primary-color);
  border: none;
  color: white;
  padding: 10px 20px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  margin: 10px;
  cursor: pointer;
  border-radius: 5px;
}

button:hover {
  background-color: #3e8e41;
}

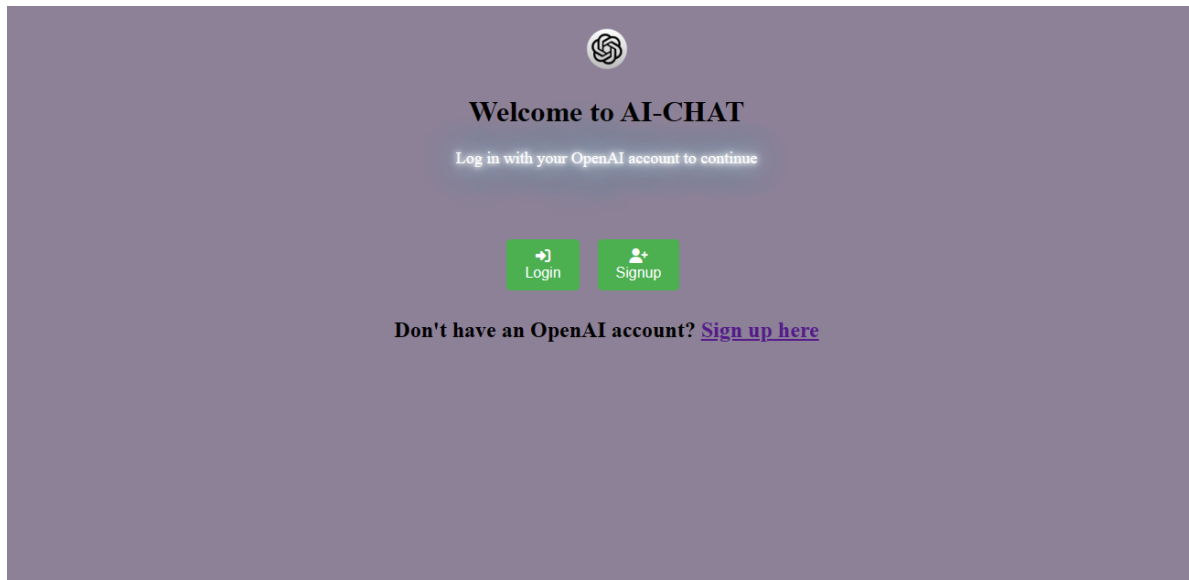
p {
  font-size: 18px;
  margin-top: 20px;
}

.container {
  display: flex;
  justify-content: space-between;
  max-width: 50px;
  margin: 0 auto;
  margin-right: 50%;
  margin-left: 35%;
}

.myimg {
  height: 50px;
  width: 50px;
}

</style>
```

Output



Signup.html

```
{% extends 'base.html' %}

{% block content %}
{% load static %}

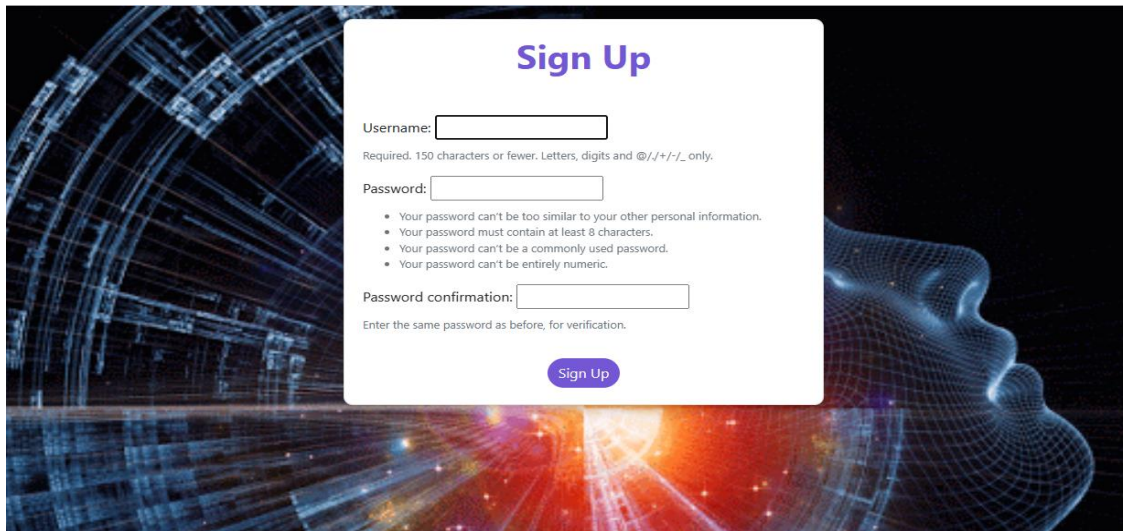
<style>
    body {
        background-image: url("{% static 'an.gif' %}");
        background-size: cover;
        background-repeat: no-repeat;
    }
    .card {
        border-radius: 10px;
        box-shadow: 0px 0px 15px 5px rgba(0,0,0,0.2);
    }
    .form-control {
        border-radius: 20px;
    }
    .btn-primary {
        background-color: #7457d4;
        border: none;
        border-radius: 20px;
    }
    .btn-primary:hover {
        background-color: #5b43ab;
```

```

    }
    h1 {
        color: #7457d4;
        font-weight: bold;
    }
</style>

<div class="container">
    <div class="row justify-content-center align-items-center vh-100">
        <div class="col-md-6">
            <div class="card">
                <div class="card-body">
                    <h1 class="text-center mb-5">Sign Up</h1>
                    <form method="post">
                        {% csrf_token %}
                        {% for field in form %}
                            <div class="form-group">
                                {{ field.label_tag }}
                                {{ field }}
                                {% if field.help_text %}
                                    <small class="form-text text-muted">{{
field.help_text }}</small>
                                {% endif %}
                            </div>
                        {% endfor %}
                        <div class="text-center">
                            <button type="submit" class="btn btn-primary mt-
3">Sign Up</button>
                        </div>
                    </form>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}

```

OutputLogin.html

```
{% extends 'base.html' %}

{% block content %}
{% load static %}
    <script src="https://kit.fontawesome.com/f3927a3561.js"
crossorigin="anonymous"></script>
    <link rel="stylesheet" href="{% static 'styles.css' %}">
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-6 col-lg-4">
            <div class="card shadow">
                <div class="card-body">
                    <h2 class="card-title text-center mb-4">Login</h2>
                    {% if messages %}
                        {% for message in messages %}
                            <div class="alert alert-{{ message.tags }}">{{ message }}</div>
                        {% endfor %}
                    {% endif %}
                    <form method="post">
                        {% csrf_token %}
                        <div class="form-group">
                            <label for="username">Username:</label>
                            <div class="input-group mb-3">
                                <span class="input-group-prepend">
                                    <span class="input-group-text">
                                        <i class="fa fa-envelope-o fa-fw"></i>
                                    </span>
                                </span>
                                <input type="text" class="form-control">
                            </div>
                        </div>
                    </form>
                </div>
            </div>
        </div>
    </div>
</div>
```

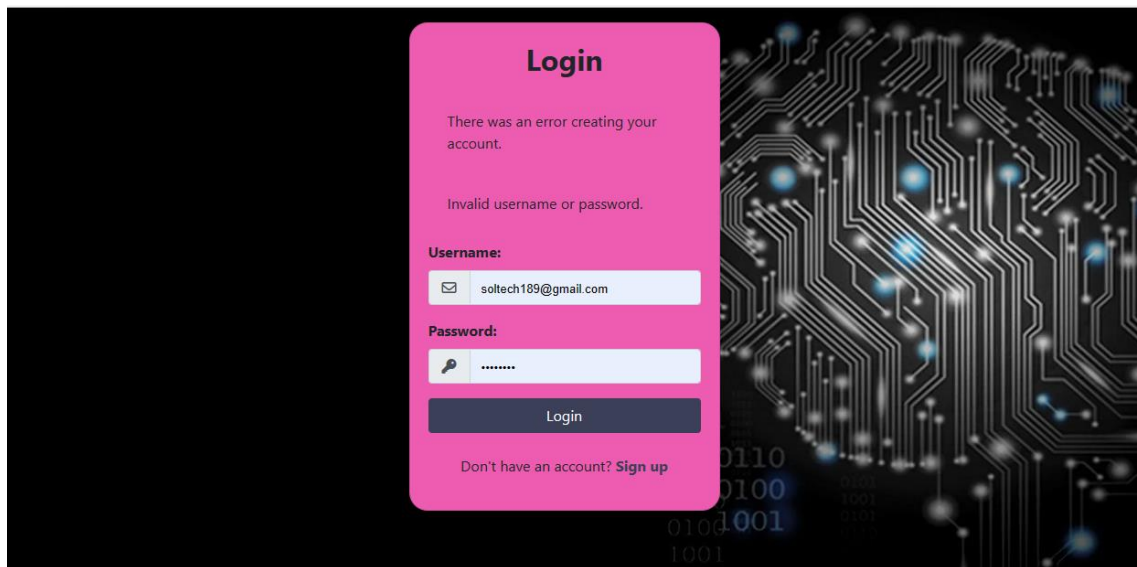

[illegible]

```

label {
  font-weight: bold;
  margin-bottom: 0.5rem;
}
.btn-primary {
  background-color: #3C3F58;
  border-color: #3C3F58;
}
.btn-primary:hover {
  background-color: #464A6E;
  border-color: #464A6E;
}
a {
  color: #3C3F58;
  font-weight: bold;
}
a:hover {
  color: #464A6E;
}
</style>
{% endblock %}

```

Output



Home.html

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Home</title>
    {% load static %}

    <script src="https://kit.fontawesome.com/f3927a3561.js"
crossorigin="anonymous"></script>
    <link rel="stylesheet" href="{% static 'styles.css' %}">
    <style>
      body {
        background-color: #043049;

      }

      h1 {
        position: relative;
        font-family: sans-serif;
        text-transform: uppercase;
        font-size: 2em;
        letter-spacing: 4px;
        overflow: hidden;
        background: linear-gradient(90deg, #000, #fff, #000);
        background-repeat: no-repeat;
        background-size: 80%;
        animation: animate 3s linear infinite;
        -webkit-background-clip: text;
        -webkit-text-fill-color: rgba(255, 255, 255, 0);
      }

      @keyframes animate {
        0% {
          background-position: -500%;
        }
        100% {
          background-position: 500%;
        }
      }
    /* Icon style */
    .fa {
      margin-right: 5px;
    }
  
```

```

/* Image style */
#chatgpt-image {
  width: 300px;
  height: 200px;
}

.my-text {
  font-size: 140%;
  color:blue;
}

.left-link {
  display: inline-block;
  background-color: #f0f0f0;
  border: 1px solid #ccc;
  border-radius: 5px;
  padding: 5px;
  float: right;
  margin-left: 10px;
  width: 600px;
  height: 800px;
  text-align: left;
  box-sizing: border-box;
}

.design{
  font-size: 30px;
  font-color:#f67dc4;
}

.login{
  font-size: 20px;
}

.signup{
  font-size: 20px;
}

.button-row {
  display: flex;
  justify-content: right;
  gap: 10px;
}

.question-image {
  height: 28px;
  width: 28px;
}

```

```

        color: white; /* Change font color to white */
    }

    .image {
        font-size: 30px;
        color: white; /* Change font color to white */
        text-decoration: none; /* Remove underline */
    }

    .button-container {
        display: flex;
        justify-content: left;
        height: 50px;
    }

    .button-row {
        display: flex;
        gap: 10px;
    }

    .button {
        background-color: #422433;
        border: none;
        color: white;
        padding: 12px 24px;
        text-align: center;
        text-decoration: none;
        display: inline-block;
        font-size: 16px;
        margin: 4px 2px;
        transition-duration: 0.4s;
        cursor: pointer;
        border-radius: 50px;
        width: 200px; /* Increase the width of the button */
        height: 50px; /* Increase the height of the button */
    }
</style>
</head>
<body>
    <a href="https://openai.com/" class="image">
    OpenAI</a>
    <div class="button-row">
    <form method="post" action="{% url 'login_view' %}">
        {% csrf_token %}

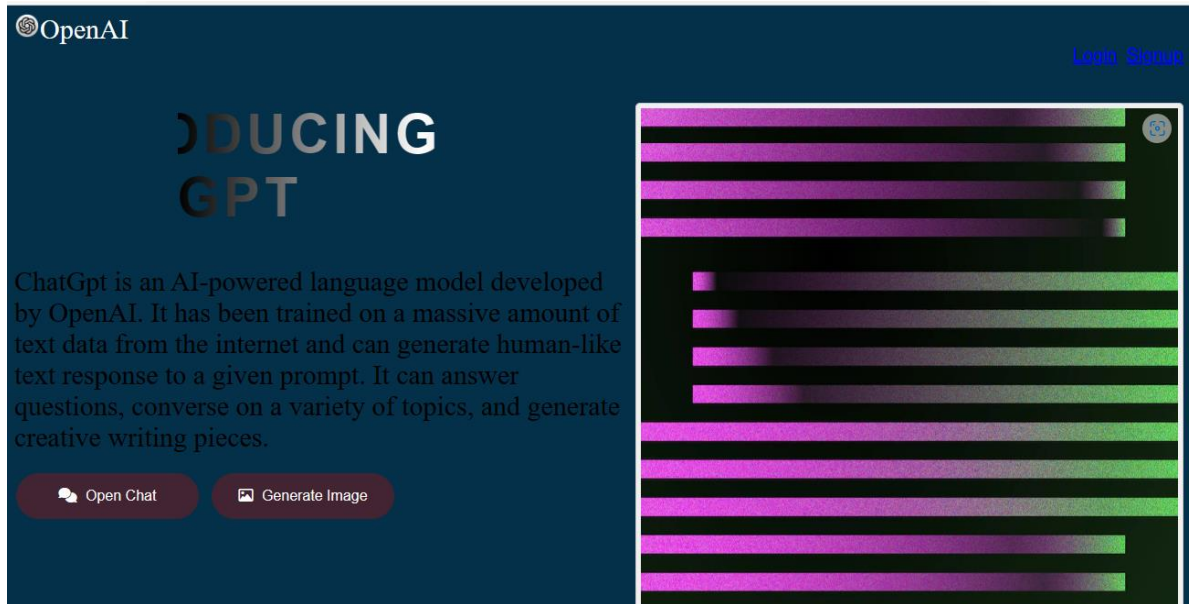
```

```

        <input type="submit" value="Login" class="login" style="background: none;
border: none; padding: 0; text-decoration: underline; color: blue; cursor:
pointer;">
    </form>
    <form method="post" action="{% url 'signup_view' %}">
        {% csrf_token %}
        <input type="submit" value="Signup" class="signup" style="background: none;
border: none; padding: 0; text-decoration: underline; color: blue; cursor:
pointer;">
    </form>
</div>
<div class="design">
    
    <h1>Introducing <br>ChatGpt</h1>
    ChatGpt is an AI-powered language model developed by OpenAI. It has been
trained on a
    massive amount of text data from the internet and can generate human-like
text response to a
    given prompt. It can answer questions, converse on a variety of topics, and
generate creative
writing pieces.
</div>
<br>
<div class="button-container">
    <div class="button-row">
        <form method="post" action="{% url 'chatgpt' %}">
            {% csrf_token %}
            <button class="button" type="submit">
                <i class="fa fa-comments"></i> Open Chat
            </button>
        </form>
        <form method="post" action="{% url 'generate_image_from_txt' %}">
            {% csrf_token %}
            <button class="button" type="submit">
                <i class="fa fa-image"></i> Generate Image
            </button>
        </form>
    </div>
</div>

</body>
</html>

```

OutputChat.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Chatbot</title>
  {% extends 'base.html' %}
  {% block content %}
    {% load static %}
    <script src="https://kit.fontawesome.com/f3927a3561.js"
crossorigin="anonymous"></script>
    <link rel="stylesheet" href="{% static 'style.css' %}">
    <script>
      function clearChat() {
        window.location.href = "{% url 'chatgpt' %}?clear=true";
      }

      function regenerateChat() {
        document.getElementById('regenerate').value = 'true';
        document.querySelector('.chat-form').submit();
      }
    </script>
    <style>
      /* Header styles */
```

```
.sms {
  background-color: #f2f2f2;
  padding: 20px;
  text-align: center;
}

.sms h1 {
  color: #333;
  font-size: 24px;
  margin: 0;
}

.new-style {
  background-color: #ddd;
  color: white;
  font-size: 16px;
  display: flex;
  justify-content: center;
  align-items: center;
  padding: 20px;
}

.new-style p {
  color: #555;
  font-size: 16px;
  margin: 0;
}
/* Rest of the CSS styles */
.message {
  margin: 10px;
  padding: 5px;
  border-radius: 5px;
  font-size: 18px;
}
.bot-message {
  background-color: #f0f0f0;
  border: 1px solid #ccc;
  text-align: left;
}

.user-message {
  background-color: #ffe6e6;
  color: rgba(106, 71, 126, 0.511);
  font-size: 20px;
}
```



```
.generated-question {
  background-color: white;
  font-size: 20px;
}

.generated-answer {
  background-color: rgba(33, 67, 58, 0.511);
  color: #643a58;
}

#button-container {
  border: 2px solid #ccc;
  width: 700px;
  text-align: left;
  background-color: #400d27;
  padding: 10px;
  border-radius: 15px;
  margin: 0 auto 10px;
}

.chat-form {
  text-align: center;
  margin-top: 10px;
}

.chat-input {
  padding: 5px;
  width: 500px;
  text-align: center;
  flex-grow: 1;
  margin-left: 10px;
  border-radius: 5px;
  border: 1px solid #ddd;
}

.input-container {
  display: flex;
  align-items: center;
  gap: 5px;
}

.regenerate-button {
  text-align: center;
  padding: 10px;
```

```
margin-left: 340px;
gap: 5px;
}
.regenerate-button {
background-color: #007bff;
color: #fff;
border: none;
padding: 10px 15px;
border-radius: 5px;
cursor: pointer;
font-size: 16px;
}

.submit-button {
padding: 5px 10px;
background-color: rgba(87, 32, 66, 0.511);
cursor: pointer;
color: #fff;
}

.question-image {
float: left;

width: 30px;
height: 30px;
align:center;
}

.answer-image {
float: left;
margin-right: 10px;
width: 30px;
height: 30px;
align: center;
}

.clear-button {
margin-top: 10px;
padding: 5px 10px;
}

/* Button styles */
.button,
.clear-button {
```

```
background-color: #422433; /* Green */
border: none;
color: white;
padding: 12px 24px;
text-align: center;
text-decoration: none;
display: inline-block;
font-size: 16px;
margin: 4px 2px;
transition-duration: 0.4s;
cursor: pointer;
border-radius: 50px;
}
```

```
/* Button hover effect */
.button:hover,
.clear-button:hover {
  background-color: #3e8e41;
}
```

```
.container {
  text-align: left;
  display: flex;
  flex-direction: column;
  align-items: flex-start;
  padding: 10px;
}
```

```
.chat-history {
  width: 300px;
  text-align: left;
  background-color: #f0f0f0;
  padding: 10px;
  border-radius: 15px;
  margin-left: 10px;
  margin-right: 10px;
  overflow-x: auto;
  overflow-y: auto;
  max-height: 900px;
  display: flex;
  flex-direction: column;
  gap: 10px;
}
```

```
.row {
  display: flex;
}

.main {
  display: flex;
  flex-direction: row;
}

.fab {
  display: flex;
}

.fab {
  display: flex;
  justify-content: flex-end;
  margin-top: 10px;
}

.fab form {
  display: flex;
  align-items: center;
  gap: 10px;
}

.fab .button {
  background-color: #422433;
  border: none;
  color: white;
  padding: 12px 24px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  transition-duration: 0.4s;
  cursor: pointer;
  border-radius: 50px;
}

.fab .button i {
  margin-right: 5px;
}

.fab .clear-button {
  background-color: #422433;
  border: none;
```

```
color: white;
padding: 12px 24px;
text-align: center;
text-decoration: none;
display: inline-block;
font-size: 16px;
transition-duration: 0.4s;
cursor: pointer;
border-radius: 50px;
}

.fab .clear-button:hover {
  background-color: #3e8e41;
}

.message {
  margin-bottom: 10px;
}

.user-message {
  background-color: #ffe6e6;
  color: rgba(106, 71, 126, 0.511);
  padding: 10px;
  border-radius: 5px;
}

.bot-message {
  background-color: #f5f5f5;
  color: #000;
  padding: 10px;
  border-radius: 5px;
}

.user-message .timestamp,
.bot-message .timestamp {
  font-size: 12px;
  color: #888;
  margin-top: 5px;
}

.timestamp::before {
  content: "[";
}
```

```
.timestamp::after {
  content: "];"
}

.user-message p,
.bot-message p {
  margin: 0;
}

.timestamp {
  margin-top: 5px;
  font-size: 12px;
  color: #888;
}

.user-message .question-image,
.bot-message .answer-image {
  width: 30px;
  height: 30px;
  margin-right: 10px;
  border-radius: 50%;
}

.bot-message .answer-image {
  width: 30px;
  height: 30px;
}

.bot-message .answer-image {
  float: left;
}

.user-message .question-image {
  float: left;
}

.clear-button {
  margin-top: 10px;
  padding: 5px 10px;
  background-color: #422433;
  border: none;
  color: white;
  font-size: 16px;
  border-radius: 50px;
  cursor: pointer;
}

.clear-button:hover {
```

```

        background-color: #3e8e41;
    }
    .scroll{
        overflow-x: auto;
        overflow-y: auto;
        max-height: 900px;
    }
</style>
<body>
<div class="row">
    <div class="sms">
        <h1>Chatbot</h1>
        <div class="new-style">
            <p class="message">We've trained a model called ChatGPT, which interacts
in a conversational way. The dialogue format makes it possible for ChatGPT to
answer follow-up questions, admit its mistakes, challenge incorrect premises, and
reject inappropriate requests.</p>
        </div>
    </div>
</div>
<div class="container">
    <div class="main">
        <div class="chat-history">
            <button class="clear-button" onclick="clearChat()">
                <i class="fa-solid fa-plus"></i>New chat</button>
            {% for conversation in conversation_history %}
                <div class="message">
                    <p class="user-message">{{ conversation.question }}</p>
                    <p class="ai-message">{{ conversation.answer }}</p>
                    <p class="timestamp">{{ conversation.timestamp }}</p>
                </div>
            {% endfor %}
        </div>
        <div class="scroll">
            <div id="button-container">
                {% for conversation in conversation_history %}
                    {% if conversation.question %}
                        <div class="message">
                            <div class="user-message {% if regenerate %}generated-question{%
endif %}">
                                
                                <p>{{ conversation.question }}</p>
                            </div>
                        </div>
                    {% endif %}
                </div>
            <div class="message">

```

```

        <div class="bot-message {% if regenerate %}generated-answer{% endif
%}">
            
            <p>{{ conversation.answer }}</p>
        </div>
    </div>
{% endfor %}

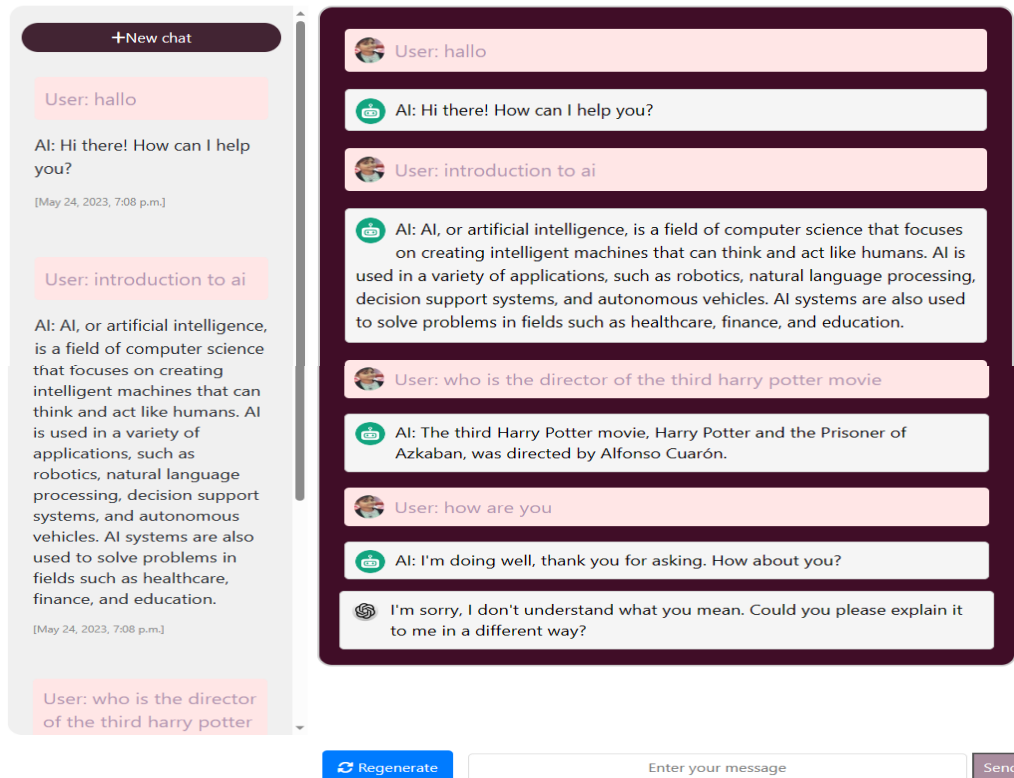
{% if chatgpt_response %}
    <div class="message bot-message">
        
        <p>{{ chatgpt_response|safe }}</p>
    </div>
{% endif %}
</div>
</div>
</div>
</div>
<form method="POST" action="{% url 'chatgpt' %}" class="chat-form">
    {% csrf_token %}
    <div class="input-container">
        <button type="button" onclick="regenerateChat()" class="regenerate-
button">
            <i class="fa-solid fa-rotate"></i> Regenerate
        </button>
        <input type="text" name="user_input" class="chat-input"
placeholder="Enter your message">
        <input type="hidden" name="context" value="{{ chatgpt_response }}">
        <input type="submit" value="Send" class="submit-button">
    </div>
</form>
<div class="fab">
    <form method="post" action="{% url 'generate_image_from_txt' %}">
        {% csrf_token %}
        <button class="button" type="submit">
            <i class="fa fa-image"></i> Generate Image
        </button>
    </form>
</div>
{% endblock content %}
</body>
</html>

```


AI-CHAT (ChatGpt)

AI-CHAT(CHATGPT)

We've trained a model called ChatGPT, which interacts in a conversational way. The dialogue format makes it possible for ChatGPT to answer follow-up questions, admit its mistakes, challenge incorrect premises, and reject inappropriate requests.



Main.html

```
{% extends 'base.html' %}

{% block content %}
{% load static %}
<script src="https://kit.fontawesome.com/f3927a3561.js"
crossorigin="anonymous"></script>
    <link rel="stylesheet" href="{% static 'styles.css' %}">
<style>
    html {
        height:100%;
    }

    body {
        margin:0;
    }

```

```

.bg {
  animation:slide 3s ease-in-out infinite alternate;
  background-image: linear-gradient(-60deg, #6c3 50%, #09f 50%);
  bottom:0;
  left:-50%;
  opacity:.5;
  position:fixed;
  right:-50%;
  top:0;
  z-index:-1;
}

.bg2 {
  animation-direction:alternate-reverse;
  animation-duration:4s;
}

.bg3 {
  animation-duration:5s;
}

.content {
  background-color:rgba(255,255,255,.8);
  border-radius:.25em;
  box-shadow:0 0 .25em rgba(0,0,0,.25);
  box-sizing:border-box;
  left:50%;
  padding:10vmin;
  position:fixed;
  text-align:center;
  top:50%;
  transform:translate(-50%, -50%);
}

h1 {
  font-family:monospace;
}

@keyframes slide {
  0% {
    transform:translateX(-25%);
  }
  100% {
    transform:translateX(25%);
  }
}

```

```

}
.container {
  max-width: 800px;
  margin: 80px auto;
  background-color: rgba(94, 28, 83, 0.511);
  border-radius: 5px;
  padding: 20px;
}
.form-group {
  display: flex;
  justify-content: center;
  align-items: center;
  margin-bottom: 20px;
}
.form-control {
  width: 60%;
  margin-right: 10px;
  border-radius: 5px;
  border: 1px solid #ccc;
  padding: 10px;
}
.btn-primary {
  background-color: #4c9bb5;
  border: none;
  border-radius: 5px;
  padding: 10px 20px;
  color: #fff;
  font-size: 16px;
  cursor: pointer;
}
.btn-primary:hover {
  background-color: #357d93;
}
.image-container {
  margin-top: 30px;
}
.image-container img {
  max-width: 100%;
  border-radius: 5px;
}
.error {
  color: red;
  margin-top: 10px;
}
.myimg {

```

```

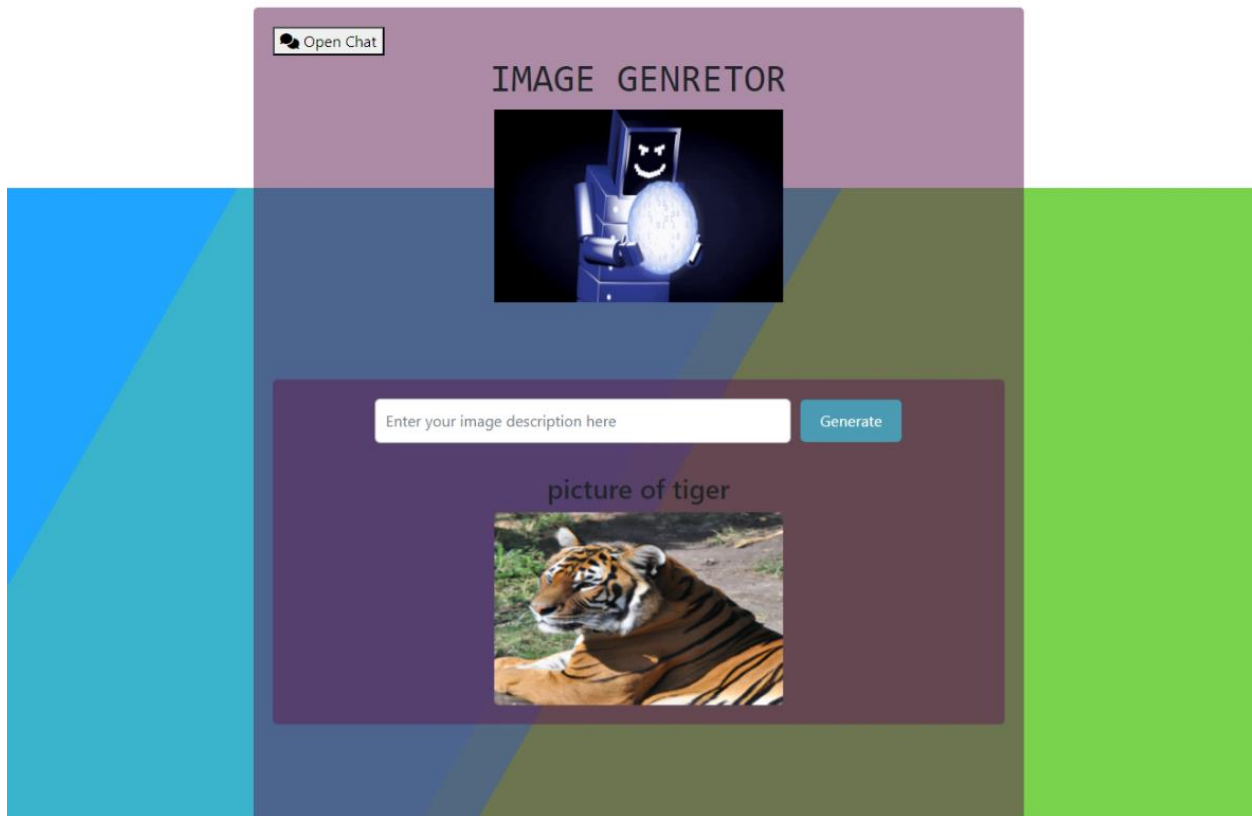
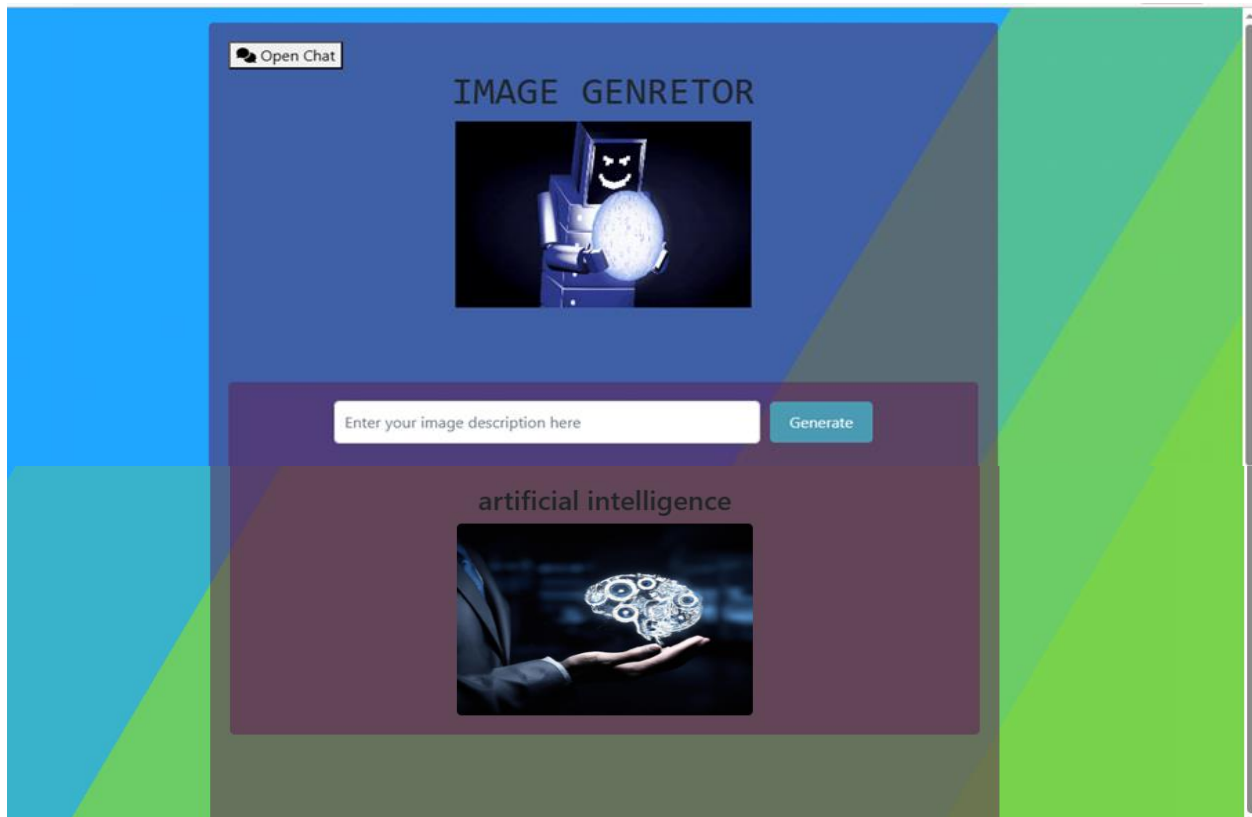
    height: 200px;
    width: 300px;
  }
</style>

<div class="button-container">
  <form method="post" action="{% url 'chatgpt' %}">
    {% csrf_token %}
    <button class="button" type="submit">
      <i class="fa fa-comments"></i> Open Chat
    </button>
  </form>
</div>
<div class="bg"></div>
<div class="bg bg2"></div>
<div class="bg bg3"></div>
<center>
<h1>IMAGE GENRETOR</h1>

<div class="container">
  <form method="POST" autocomplete="off">
    {% csrf_token %}
    <div class="form-group">
      <input type="text" class="form-control" name="user_input"
placeholder="Enter your image description here" required>
      <button type="submit" class="btn btn-primary">Generate</button>
    </div>
  </form>
  {% if object %}
    <div class="image-container">
      {% if object.error %}
        <div class="error">{{ object.error }}</div>
      {% else %}
        <h3>{{ object.phrase }}</h3>
        
        {% endif %}
      </div>
    {% endif %}
  </div>
  {% endblock content %}

```

OUTPUT



4.4 STATICFILES

The staticfiles framework is responsible for managing static files such as **CSS**, **JavaScript**, **images**, and other assets. The staticfiles framework provides a convenient way to collect, store, and serve these static files.

Here's an explanation of staticfiles in Django:

Purpose:

- Static files are files that are served directly to the client without any processing by Django.
- The staticfiles framework helps manage these files and ensures they are easily accessible during development and deployment.

Configuration:

- To use the staticfiles framework, you need to configure certain settings in your project's settings.py file.
- The `STATIC_URL` setting specifies the base URL for serving static files.
- The `STATIC_ROOT` setting defines the directory where collected static files will be stored during deployment.

Static Files Directory:

- In each application, you can create a static directory to store application-specific static files.
- By default, Django will automatically discover and collect static files from all installed applications.

Static File Storage:

- During development, the staticfiles framework can serve static files directly from the `STATICFILES_DIRS` or application-specific static directories.
- During deployment, you need to collect static files into a single directory using the `collectstatic` management command.
- Collected static files are stored in the `STATIC_ROOT` directory and should be served by a web server or a file-serving mechanism.

Static File URLs:

- In templates, you can reference static files using the `{% static %}` template tag.
- The `{% static %}` tag generates the URL for a static file based on the `STATIC_URL` setting.
- Development vs. Production:
 - In development mode (`DEBUG=True`), Django automatically serves static files from the `STATICFILES_DIRS` and application-specific static directories.
 - In production mode (`DEBUG=False`), you need to configure your web server or file-serving mechanism to serve static files from the `STATIC_ROOT` directory.

- The staticfiles framework simplifies the management of static files in a Django project. It allows you to organize and collect static files from different applications into a single location for easy deployment.

Properly configure the **STATIC_URL**, **STATIC_ROOT**, and **STATICFILES_DIRS** settings in project's **settings.py** file to ensure static files are served correctly.

SETTING.PY

```

12 from pathlib import Path
13 # Build paths inside the project like
14 BASE_DIR = Path(__file__).resolve().parent
15 # Quick-start development settings - see
16 # See https://docs.djangoproject.com/en/4.2/ref/settings/
17
18 # SECURITY WARNING: keep the secret key
19 SECRET_KEY = 'django-insecure-u0%a%qi'
20
21 # SECURITY WARNING: don't run with debug
22 DEBUG = True
23
24 ALLOWED_HOSTS = ["*"]
25 import os
26
27 DATABASES = {
28     'default': {
29         'ENGINE': 'django.db.backends.sqlite3',
30         'NAME': BASE_DIR / 'db.sqlite3',
31     }
32 }

```

default database

STATICFILES_DIRS, STATIC_URL, MEDIA_ROOT PATH SETUP(setting.py)

TEMPLATES(setting.py)

```

122 import os
123 STATICFILES_DIRS=[
124     os.path.join(BASE_DIR, 'static')
125 ]
126
127
128 STATIC_URL = '/static/'
129 MEDIA_ROOT=os.path.join(os.path.dirname(BASE_DIR), 'media')
130 # Default primary key field type
131 # https://docs.djangoproject.com/en/4.2/ref/settings/#default-auto-field
132
133 DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
134

```

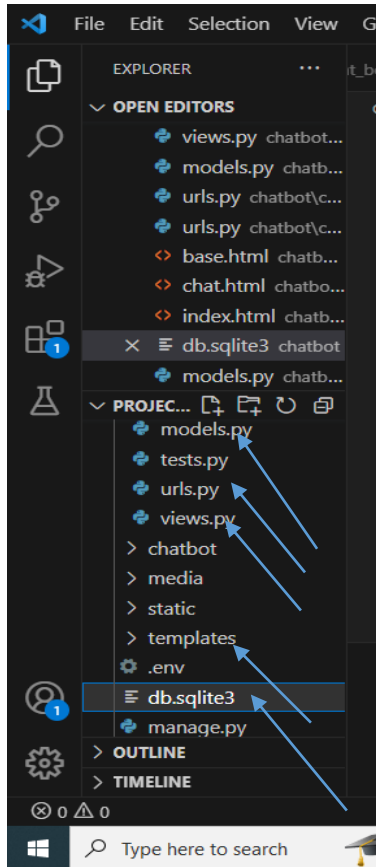
CHAPTER 5

DATABASE

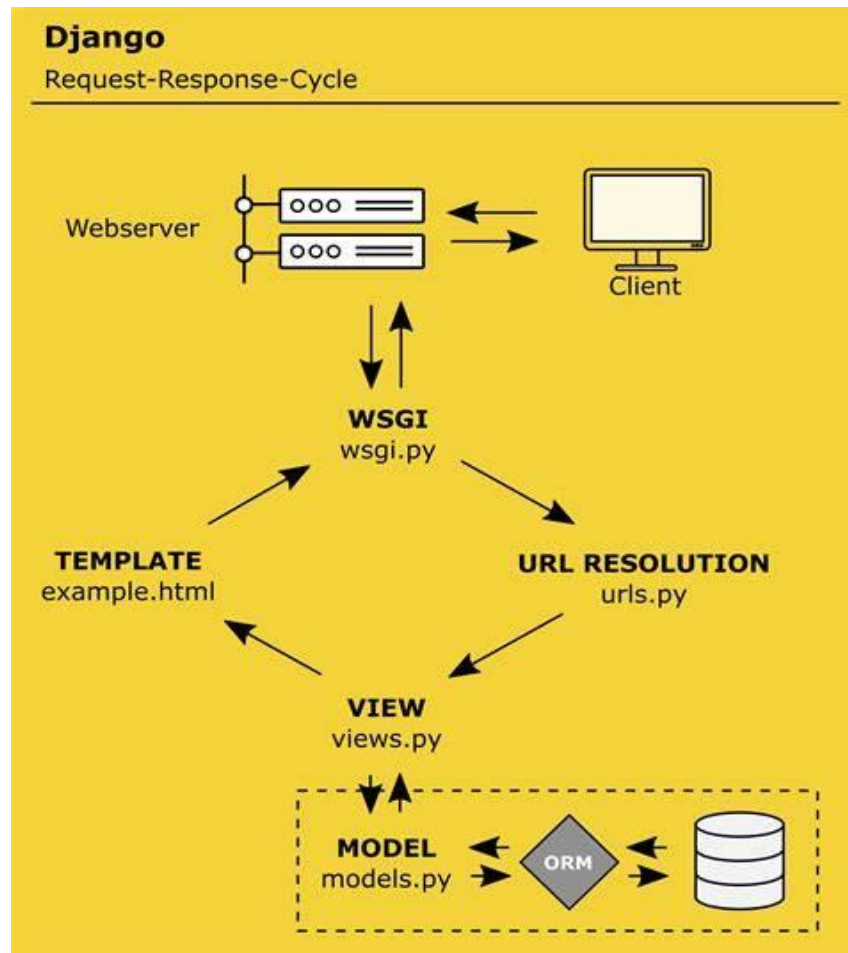
When designing a database for a Django project like ChatGPT, need to consider the specific requirements of your application and how the data will be structured and stored. Here's a brief overview of the database design considerations for ChatGPT in Django:

1. User Model:
 - Create a User model that represents the users of the ChatGPT application.
 - The User model can be created by extending Django's built-in `AbstractUser` or `AbstractBaseUser` classes.
 - Include fields such as username, email, password, and any additional user-specific data.
2. Message Model:
 - Create a Message model to store the messages exchanged in the chat.
 - The Message model can have fields like sender (ForeignKey to User model), recipient (ForeignKey to User model), content, timestamp, etc.
 - Additional fields such as read/unread status, message type, or attachments can also be included based on your requirements.
3. Room/Conversation Model (optional):
 - If ChatGPT supports multiple chat rooms or conversations, you can create a Room or Conversation model.
 - This model can have fields like participants (ManyToManyField to User model), title, creation timestamp, etc.
 - The Message model can then have a ForeignKey to the Room/Conversation model to associate messages with a specific chat.
4. Database Relationships:
 - Establish relationships between models as necessary.
 - For example, the User model can have a one-to-many relationship with the Message model (one user sending multiple messages).
 - If a Room/Conversation model is included, it can have a many-to-many relationship with the User model (multiple users participating in a room).
5. Database Optimization:
 - Consider optimizing database queries based on your application's needs.
 - Use Django's ORM features like `select_related()` or `prefetch_related()` to reduce database queries when fetching related objects.
 - Implement appropriate indexes and constraints on fields for efficient data retrieval and integrity.
6. Database Migration:

- Once you have defined your models, use Django's migration system to create the corresponding database tables and schema.
- Django's migration commands (`makemigrations` and `migrate`) will handle the creation and modification of database tables based on your model definitions.



Models.py



```

from django.db import models
from datetime import datetime
class ChatConversation(models.Model):
    question = models.TextField()
    answer = models.TextField()
    timestamp = models.DateTimeField(default=datetime.now)
def __str__(self):
    return f"Question: {self.question}, Answer: {self.answer}"
  
```

CHAT DATABASE TABLE

on

The screenshot shows the SQLiteOnline.com interface for a database named 'db.sqlite3'. The left sidebar lists the database schema, including tables like 'auth_group', 'auth_group_permissions', 'auth_permission', 'auth_user', 'auth_user_groups', 'auth_user_user_permissions', 'chat_bot_chatconversation', 'chat_bot_image', 'django_admin_log', 'django_content_type', 'django_migrations', 'django_session', and 'sqlite_sequence'. The 'chat_bot_chatconversation' table is selected, and its columns are listed: 'id' (INTEGER), 'question' (TEXT), 'answer' (TEXT), and 'timestamp' (datetime). The main pane shows the SQL query 'SELECT * FROM chat_bot_chatconversation' and a table of results with 7 rows (IDs 260-266).

id	question	answer	timestamp
260	User: who is the di...	AI: The director of ...	2023-05-19 17:17:26.58...
261	User: introduction t...	AI: I'm sorry, I don'...	2023-05-19 17:17:43.65...
262	User: how are you	AI: I'm doing well, t...	2023-05-19 17:18:02.41...
263	User: who is the di...	AI: The director of ...	2023-05-19 17:18:18.83...
264	User: write a progr...	AI: #include <stdio...	2023-05-19 17:19:20.11...
265	User: write a progr...	AI: To write a progr...	2023-05-19 19:49:42.49...
266	User: write a progr...	AI: Here is an exa...	2023-05-19 19:51:26.09...

Images MODELS.PY DATABASE TABLE

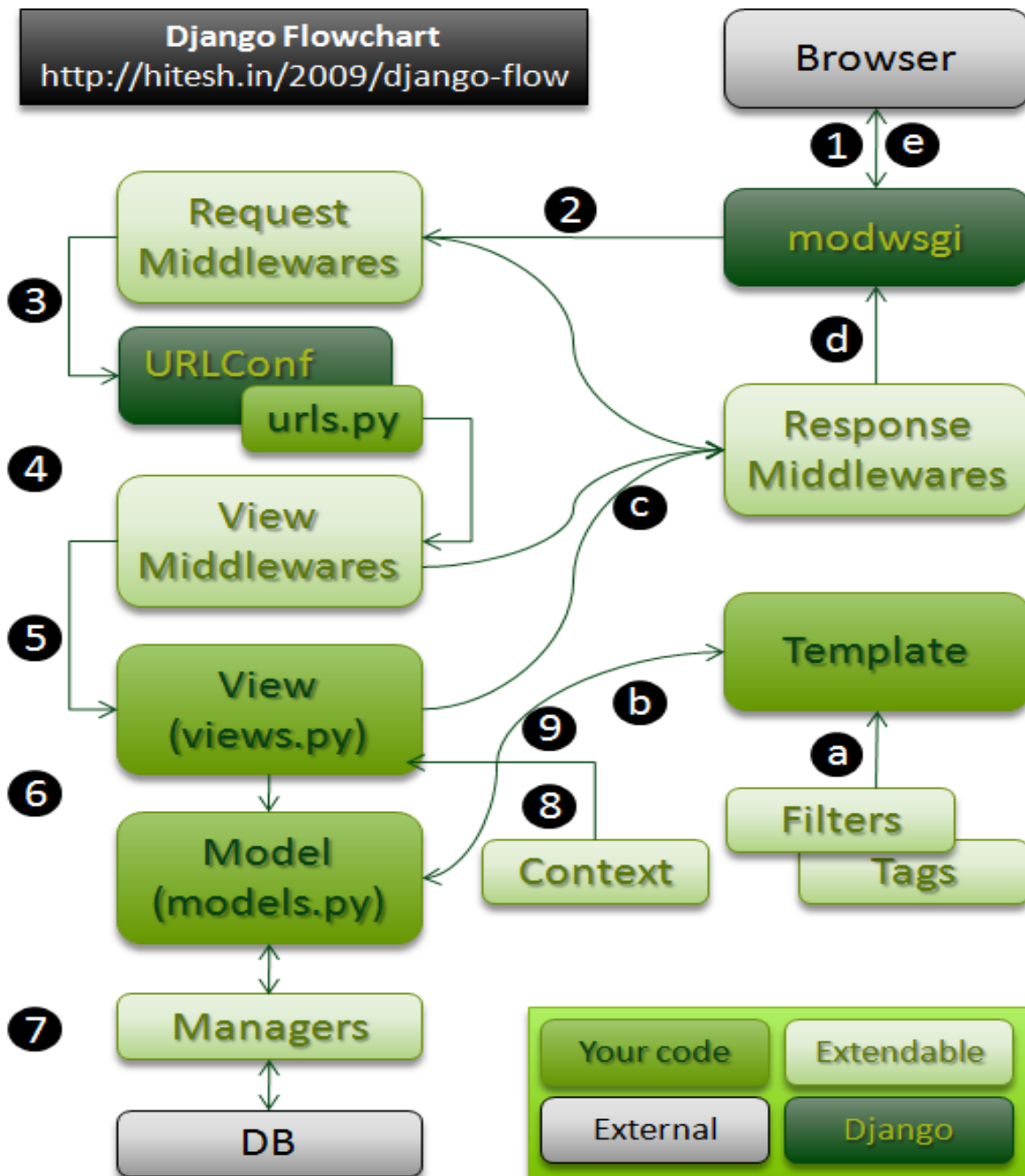
The screenshot shows the SQLiteOnline.com interface for a database named 'db.sqlite3'. The left sidebar lists the database schema, including tables like 'auth_permission', 'auth_user', 'auth_user_groups', 'auth_user_user_permissions', 'chat_bot_chatconversation', 'chat_bot_image', 'django_admin_log', 'django_content_type', 'django_migrations', 'django_session', and 'sqlite_sequence'. The 'chat_bot_chatconversation' table is selected, and its columns are listed: 'id' (INTEGER), 'phrase' (varchar(200)), and 'ai_image' (varchar(100)). The main pane shows the SQL query 'SELECT * FROM chat_bot_chatconversation' and a table of results with 7 rows (IDs 260-266). Overlaid on the right is a Django model definition for 'Image'.

```

from django.db import models
class Image(models.Model):
    phrase = models.CharField(max_length=200)
    ai_image = models.ImageField(upload_to='chatbot')
    def __str__(self):
        return str(self.phrase)

```

id	question	answer	timestamp
260	User: who is the di...	AI: The director of ...	2023-05-19 17:17:26.58...
261	User: introduction t...	AI: I'm sorry, I don'...	2023-05-19 17:17:43.65...
262	User: how are you	AI: I'm doing well, t...	2023-05-19 17:18:02.41...
263	User: who is the di...	AI: The director of ...	2023-05-19 17:18:18.83...
264	User: write a progr...	AI: #include <stdio...	2023-05-19 17:19:20.11...
265	User: write a progr...	AI: To write a progr...	2023-05-19 19:49:42.49...
266	User: write a progr...	AI: Here is an exa...	2023-05-19 19:51:26.09...

ACTUAL WORK PROJECT

(Fig:4 Data Flowgram actual work project)

CHAPTER 6

USER AUTHENTICATION AND AUTHORIZATION

User authentication and authorization are essential aspects of web applications, including those built with Django. They ensure that only authorized users can access certain features or perform specific actions within the application. Here's an explanation of user authentication and authorization in Django:

Authentication:

- Authentication verifies the identity of users accessing the application.
- Django provides a built-in authentication system that handles user authentication and manages user sessions.
- Users can create accounts, log in, log out, and reset passwords through Django's authentication views and forms.
- Authentication in Django involves the use of user models, authentication backends, authentication views, and authentication middleware.

User Models:

- Django provides a default user model (`django.contrib.auth.models.User`) that includes basic fields such as **username**, **password**, **email**, etc.
- You can customize the user model by either extending the default model or creating a completely custom user model.

Authentication Backends:

- Django supports multiple authentication backends, allowing you to authenticate users using different methods, such as **username/password**, **email/password**, **social authentication (OAuth)**, etc.
- You can configure the authentication backend(s) in the project's settings to specify the order and type of authentication methods used.

Authentication Views and Forms:

Django provides pre-built views and forms for common authentication actions, such as login, logout, password reset, etc.

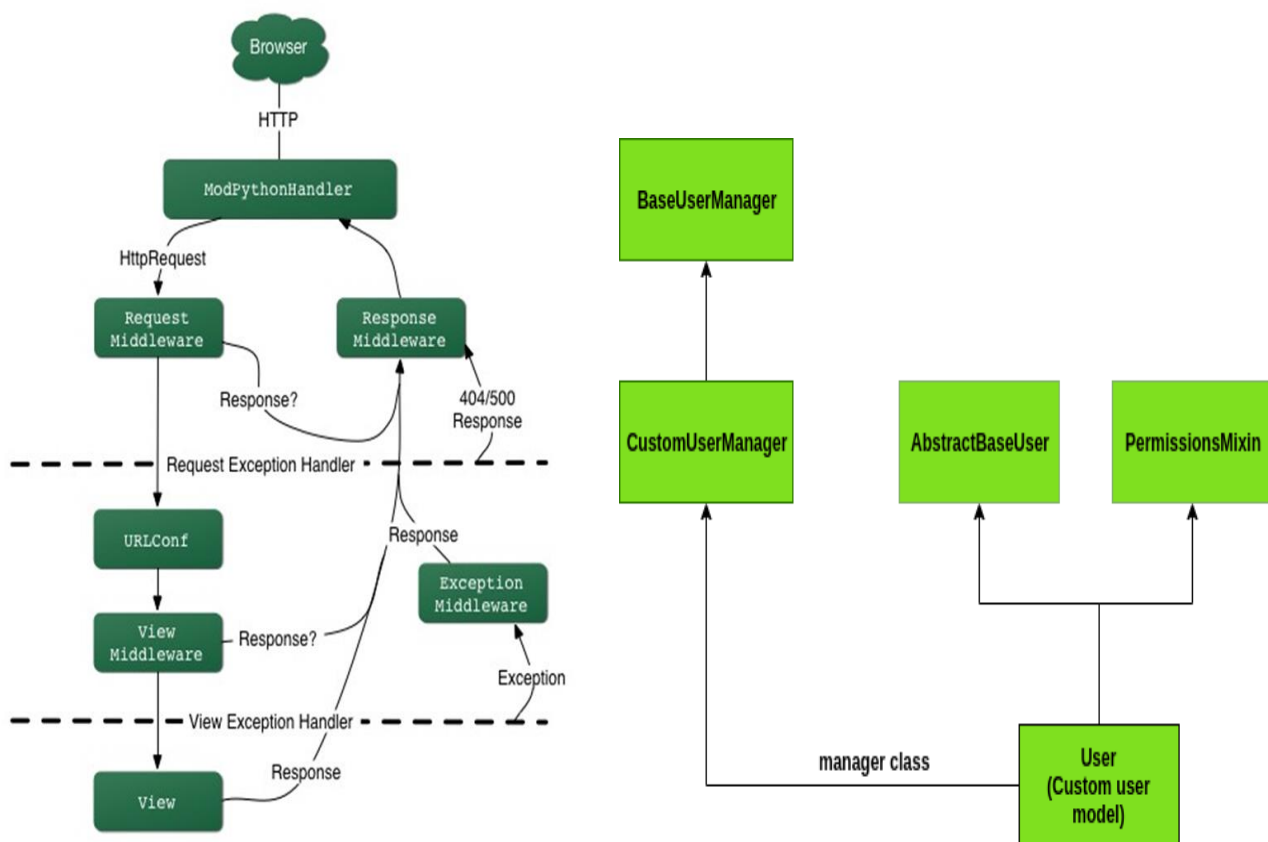
These views handle user input, validate credentials, and manage user sessions.

Authorization:

- **Authorization** determines the permissions and access rights of authenticated users.
- Django provides a flexible and granular permission system that allows you to define access control at the model, view, or object level.
- Permissions can be assigned to user groups or individual users, and they can be checked using decorators, **mixins**, or in view logic.

Decorators and Mixins:

- Django provides decorators like **@login_required** that can be applied to views to ensure that only authenticated users can access them.
- Mixins like **LoginRequiredMixin** can be used with class-based views to enforce authentication requirements.
- User authentication and authorization are crucial for protecting sensitive information, restricting access to specific functionality, and ensuring the security of your application. By utilizing Django's built-in authentication system and permission framework, you can implement robust user management and access control in your Django project.



(Fig:5 USER AUTHENTICATION AND AUTHORIZATION)

CHAPTER 7

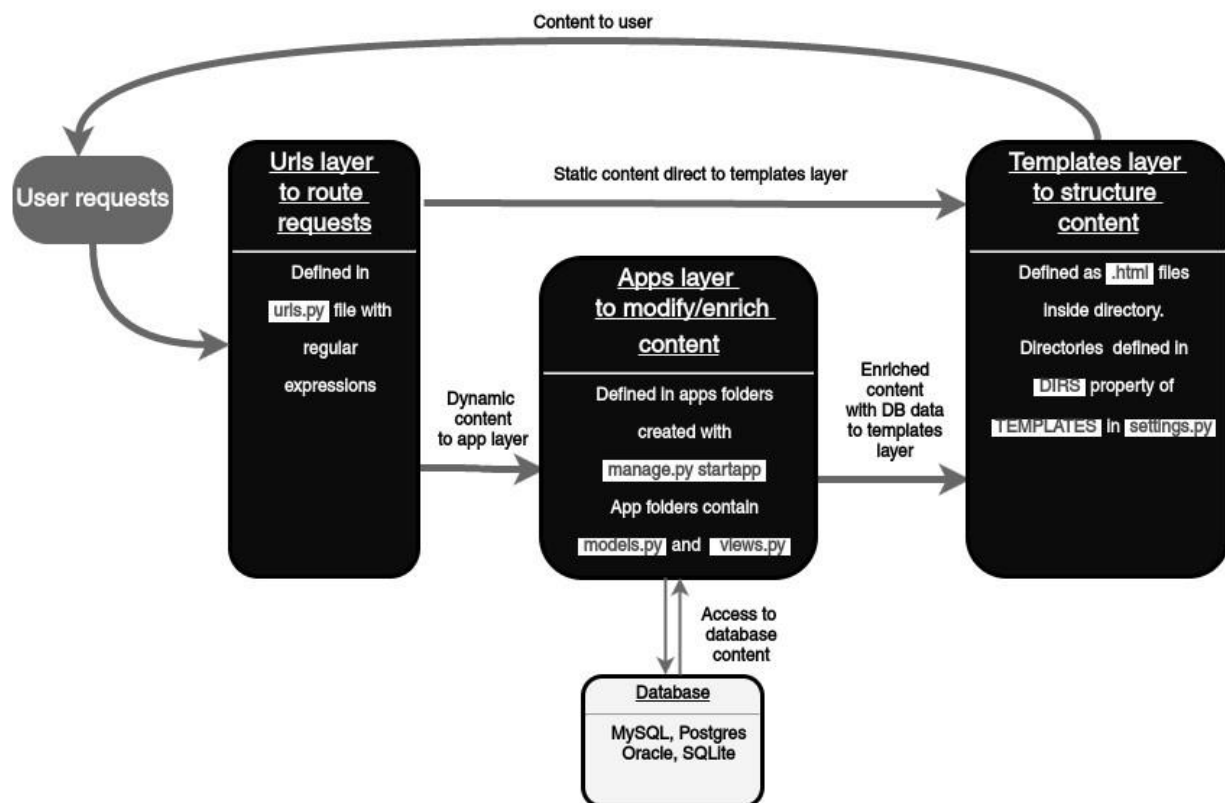
KEY FEATURES AND FUNCTIONALITY

The key features and functionality of ChatGPT implemented using Django can be summarized as follows:

1. User Registration and Authentication:
 - Users can register for a new account and authenticate themselves using their credentials.
 - Django's built-in authentication system handles user registration, login, and password management.
2. Real-time Chat Interface:
 - ChatGPT provides a real-time chat interface where users can engage in conversations.
 - The chat interface allows users to send and receive messages in real-time, providing an interactive and dynamic user experience.
3. Multi-user Chat Rooms or Private Conversations:
 - ChatGPT supports multiple chat rooms or private conversations among users.
 - Users can create or join existing chat rooms to communicate with multiple participants simultaneously.
 - Private conversations can be initiated between two or more specific users, enabling more personalized interactions.
4. Message Storage and Retrieval:
 - ChatGPT stores and retrieves chat messages, allowing users to view past conversations.
 - Messages are associated with users and organized in a database for easy retrieval and display.
5. User Profiles and Settings:
 - Users can customize their profiles and update their personal settings.
 - Profile customization may include uploading a profile picture, providing a bio, or specifying personal preferences.
6. Security and Authorization:
 - Django's authentication and authorization mechanisms ensure that only authenticated users can access ChatGPT.
 - Users may have different roles or permissions, allowing for restricted access to certain features or actions.
7. Error Handling and Validation:
 - ChatGPT performs appropriate error handling and validation of user input to ensure data integrity and prevent security vulnerabilities.
 - Django's form validation and error reporting mechanisms help maintain the reliability of the application.
8. Deployment and Scalability:

- ChatGPT can be deployed on various platforms, making it accessible to a wide range of users.
- Django's scalability features, such as caching and load balancing, can be utilized to handle increased user traffic and ensure optimal performance.

The above features and functionality provide a foundation for building a robust and interactive chat application using Django and integrating it with ChatGPT. It allows users to engage in real-time conversations, create chat rooms, and personalize their experience within a secure and user-friendly environment.



(Fig: KEY FEATURES AND FUNCTIONALITY)

CHAPTER 8

APIS AND WEB SERVICES

In a ChatGPT application implemented using Django, you can utilize APIs and web services to enhance functionality and integrate external services. Here are some common use cases for APIs and web services in ChatGPT with Django:

1. Natural Language Processing (NLP) APIs:

- Integrate NLP APIs such as Google Cloud Natural Language API, IBM Watson NLU, or spaCy to perform language processing tasks like sentiment analysis, entity recognition, or text classification.
- These APIs can enhance the understanding and analysis of user messages within the chat application.

2. Chatbot APIs:

- Incorporate pre-trained chatbot APIs like Dialogflow, Wit.ai, or Microsoft Bot Framework to enhance the conversational capabilities of ChatGPT.
- These APIs provide advanced natural language understanding and response generation to create more interactive and intelligent conversations.

3. External Data Retrieval:

- Integrate APIs to retrieve data from external sources and provide real-time information to users in the chat application.
- For example, you could integrate weather APIs to provide weather forecasts, news APIs to deliver news updates, or financial APIs to fetch stock prices.

4. User Authentication and Authorization:

- Utilize authentication and authorization APIs like OAuth, JWT, or social media login APIs (e.g., Google, Facebook, Twitter) to enable seamless user authentication and registration.
- These APIs allow users to log in with existing accounts and provide secure access to the chat application.

5. Content Generation or Translation APIs:

- Use APIs like OpenAI's GPT-3 or Google Translate to generate content or facilitate real-time translation within the chat application.
- These APIs can help generate responses, provide language translations, or support multilingual conversations.

6. File Storage and Retrieval:

- Integrate cloud storage APIs such as Amazon S3, Google Cloud Storage, or Dropbox to allow users to upload and share files within the chat application.
- These APIs enable efficient storage and retrieval of user-generated content like images, documents, or media files.

7. Webhooks and Web Services Integration:

- Implement webhooks to receive notifications or trigger actions from external services based on specific events within the chat application.
- Webhooks can be used to integrate with third-party services like email notifications, task management tools, or analytics platforms.

8. Payment Gateways and E-commerce APIs:

- Integrate payment gateways and e-commerce APIs (e.g., PayPal, Stripe) to enable secure online transactions and facilitate in-chat purchases or subscriptions.

By leveraging APIs and web services, you can extend the functionality of ChatGPT and integrate external services seamlessly. Django's flexible architecture and support for RESTful APIs make it well-suited for integrating various APIs and web services into your chat application.



(Fig: APIS AND WEB SERVICE)

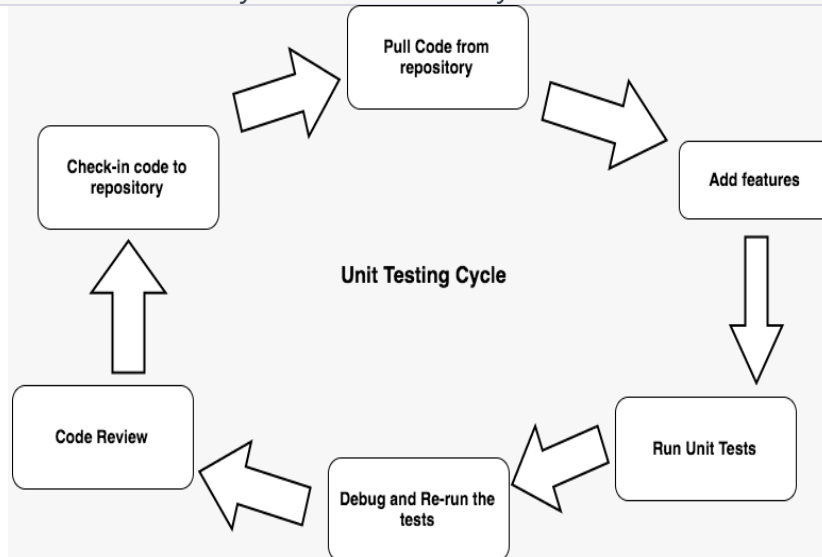
CHAPTER 9

TESTING AND QUALITY ASSURANCE

Testing and quality assurance are critical aspects of the software development process to ensure that your web application or API meets the required quality standards. Here's a brief description of testing and quality assurance:

1. Testing Types:

- **Unit Testing:** Unit testing focuses on testing individual components or units of code in isolation to verify their functionality.

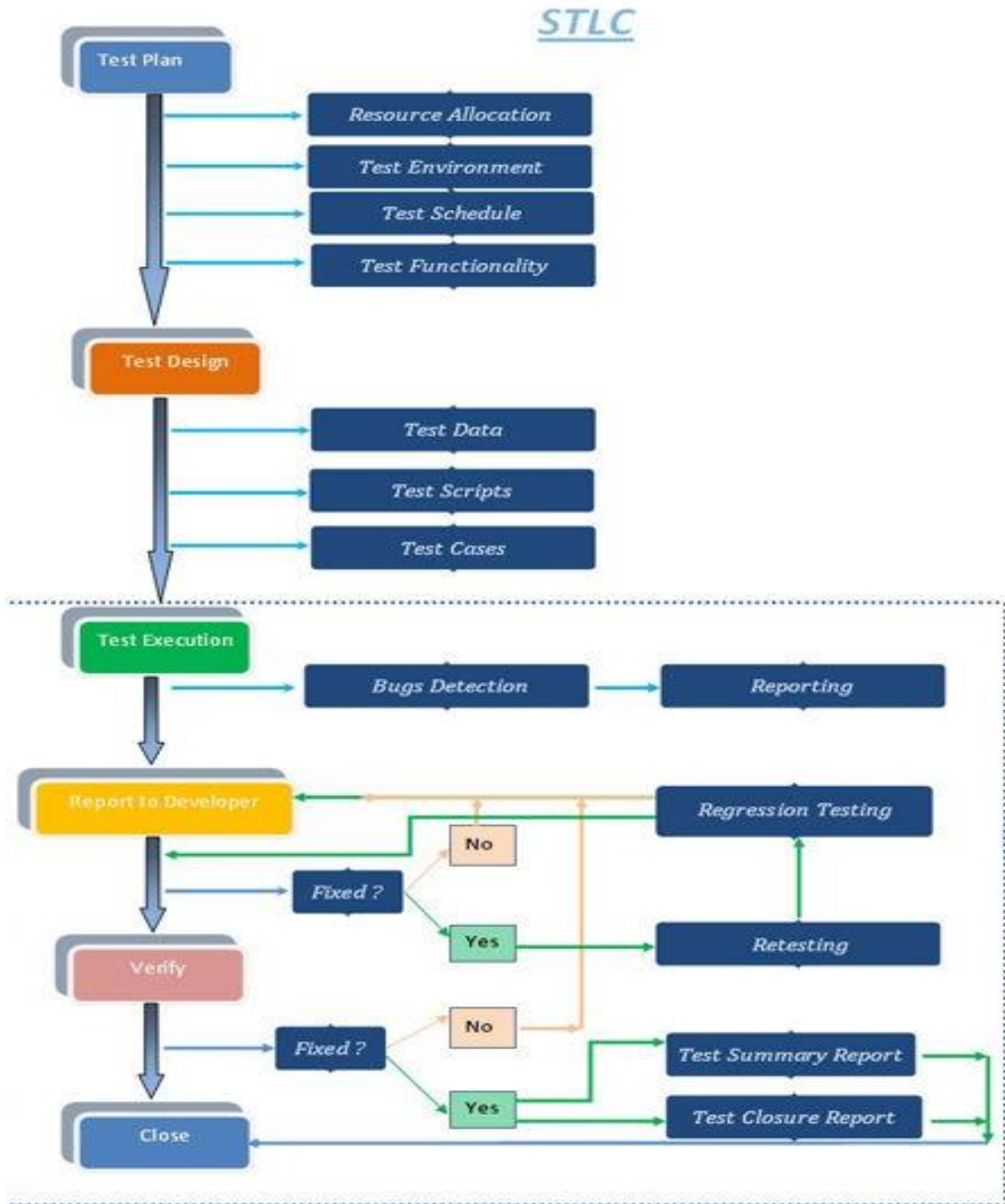


- **Integration Testing:** Integration testing checks the interactions and compatibility between different components, modules, or systems to ensure they work together correctly.
- **Functional Testing:** Functional testing validates that the application or API meets the specified functional requirements and performs as expected.
- **Performance Testing:** Performance testing assesses the responsiveness, scalability, and stability of the application under various load conditions.
- **Security Testing:** Security testing identifies vulnerabilities and ensures the application or API is protected against potential security threats.
- **User Acceptance Testing (UAT):** UAT involves testing the application with end users to verify if it meets their requirements and expectations.

2. Test Automation:

- Test automation involves using tools and frameworks to automate the execution of tests, making the testing process faster and more efficient.
- Automated tests can be created for unit testing, functional testing, regression testing, and other types of testing.

- Popular test automation frameworks for web applications include Selenium, Cypress, and Puppeteer.
- 3. Continuous Integration and Continuous Delivery/Deployment (CI/CD):
 - CI/CD is a set of practices that enable automated building, testing, and deployment of applications.
 - CI involves automatically integrating code changes into a shared repository and running tests to detect issues early.
 - CD automates the process of deploying applications to production environments after passing tests and meeting quality criteria.
- 4. Bug Tracking and Issue Management:
 - Bug tracking systems, such as JIRA or Bugzilla, help track and manage reported bugs and issues throughout the development lifecycle.
 - These systems facilitate communication and collaboration between developers, testers, and stakeholders for efficient issue resolution.
- 5. Code Reviews and Quality Metrics:
 - Code reviews involve peer review of code changes to identify issues, ensure code quality, and share knowledge among team members.
 - Static code analysis tools can be used to identify potential code quality issues and enforce coding standards.
- 6. Regression Testing:
 - Regression testing ensures that existing functionalities continue to work as expected after making changes or adding new features.
 - Automated regression testing can be performed using test suites that cover critical areas of the application. By incorporating testing and quality assurance practices, you can identify and address issues early in the development cycle, improve software quality, and deliver a reliable and user-friendly web application or API.



(fig: flow chart Testing)

CHAPTER 10

DEVELOPMENT ENVIRONMENT AND TECHNOLOGIES

- To develop a Django-based project for ChatGPT, you would require the following development environment and technologies:

I. **Development Environment:**

- Operating System: Choose an operating system of your preference, such as **Windows, macOS, or Linux**.
- Python: Install Python, which is the programming language used for Django development. Make sure you have the latest version of Python installed on your system.

II. **Django Framework:**

- Django: Django is a high-level Python web framework that provides a robust set of tools and libraries for building web applications.
- Install Django using pip, the Python package manager, and set up a new Django project using the command-line interface.

III. **Database:**

- Choose a database management system (DBMS) that Django supports. Popular choices include **SQLite** (for development), PostgreSQL, MySQL, or Oracle.
- Install and configure the DBMS of your choice and specify the database settings in the Django project's configuration file (settings.py).

IV. **Additional Django Libraries and Tools:**

- Django REST Framework: If you intend to build a RESTful API for ChatGPT, you can utilize Django REST Framework (DRF), a powerful toolkit for building APIs.
- Django Channels: If you plan to incorporate real-time functionality, you can use Django Channels, which provides a WebSocket framework for Django.
- Django Templating Engine: Django has its own templating engine for rendering HTML templates, which you can use for frontend development.

V. **ChatGPT Integration:**

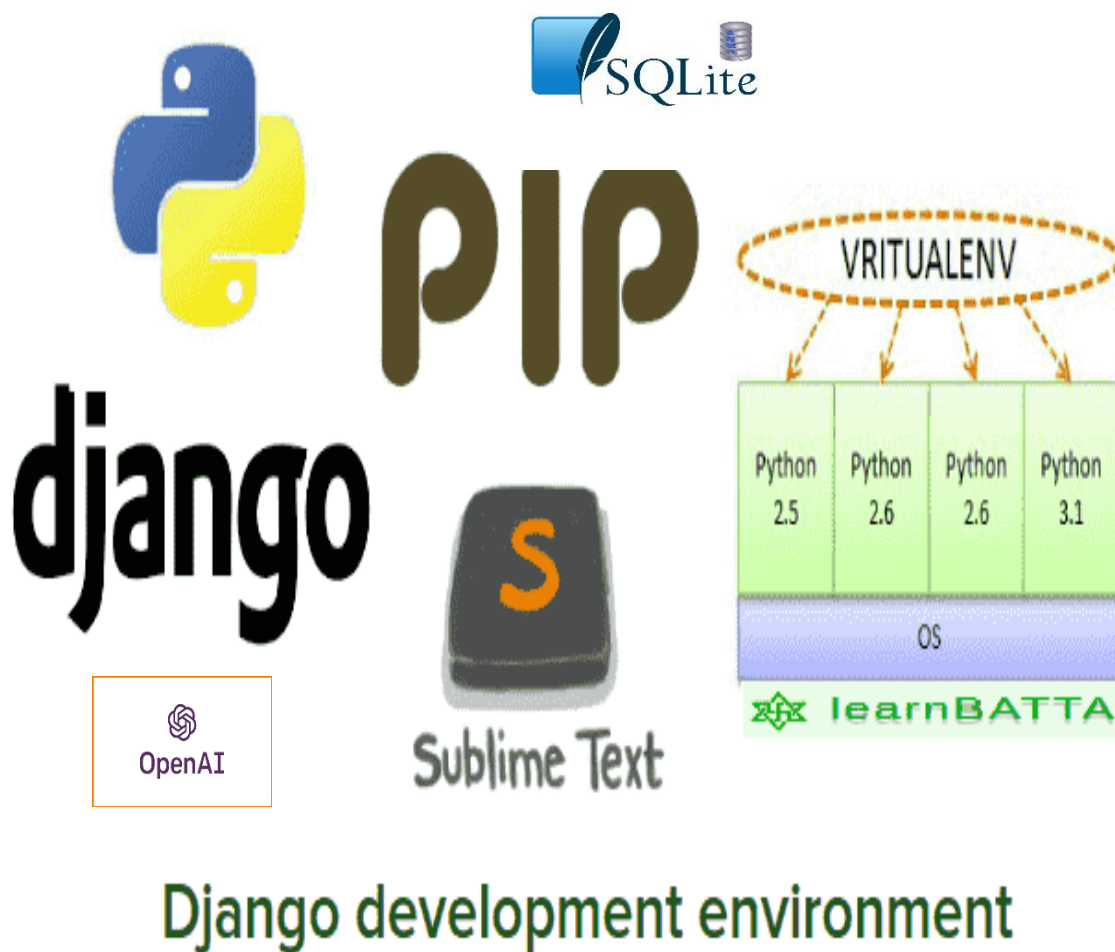
- OpenAI API: To integrate ChatGPT into Django project, you will need access to the **OpenAI API**. Follow OpenAI's documentation to obtain an **API key** and understand how to make API requests.

- Python HTTP Libraries: Utilize Python libraries such as requests or **http.client** to make HTTP requests to the **OpenAI API** and retrieve responses.

VI. Development Tools:

- Integrated Development Environment (IDE): Choose an IDE or code editor of your choice, such as **PyCharm**, **Visual Studio Code**, or **Sublime Text**, to facilitate coding and development.
- Version Control: Use a version control system like Git to track changes, collaborate with others, and manage your codebase effect

○



(Fig: Django development environment)

CHAPTER 11

FUTURE PLANS AND ENHANCEMENTS

Future plans and enhancements for ChatGPT using Django can include various aspects to improve the application's functionality, performance, and user experience. Here are some potential ideas for future development:

1. Enhanced **Natural Language Processing** (NLP):
 - Explore advanced NLP techniques to improve the understanding and response generation capabilities of **ChatGPT**.
 - Experiment with state-of-the-art models, such as **GPT-3** or future iterations, to leverage their enhanced language generation capabilities.
2. Integration with External Services:
 - Integrate ChatGPT with external services and APIs to provide additional functionality and data sources.
 - Incorporate third-party APIs for tasks like **language translation, sentiment analysis, or image recognition** to enrich the user experience.
3. User Management and Personalization:
 - Implement user management features to allow users to have personalized profiles, preferences, and historical conversations.
 - Provide user-specific recommendations or suggestions based on their interaction history.
4. Multi-Channel Support:
 - Extend the application to support multiple communication channels, such as web chat, mobile apps, social media platforms, or voice assistants.
 - Implement **APIs or webhooks** to enable integration with popular messaging platforms like Slack, WhatsApp, or Facebook Messenger.
5. Contextual Understanding and Conversation Flow:
 - Enhance the chatbot's ability to maintain context and continuity in conversations.
 - Develop mechanisms to track conversation history and use it to provide more coherent and relevant responses.
6. Improved Security and Privacy:
 - Implement security measures to protect user data and ensure secure communication channels.
 - Apply encryption techniques and best practices to safeguard sensitive information exchanged during conversations.
7. Performance Optimization:
 - Fine-tune the application for better performance and scalability, considering factors such as response time, server load, and concurrent user handling.

- Employ caching mechanisms, query optimization, or asynchronous processing to optimize the application's performance.
8. Analytics and Reporting:
 - Integrate analytics tools to collect data on user interactions, conversation patterns, and user satisfaction.
 - Use these insights to improve the chatbot's performance, identify areas for enhancement, and make data-driven decisions.
 9. Continuous Testing and Deployment:
 - Implement automated testing frameworks and continuous integration/continuous deployment (CI/CD) pipelines to ensure high-quality releases and quick deployment cycles.
 - Set up automated regression tests to catch potential issues with new releases.
 10. User Feedback and Iterative Development:
 - Collect user feedback through surveys, ratings, or feedback forms to understand user satisfaction and identify areas for improvement.
 - Regularly iterate and improve the chatbot based on user feedback and usage patterns.

These are just some potential future plans and enhancements for ChatGPT using Django. The specific roadmap will depend on your project's goals, user needs, and available resources.



CHAPTER 12

FALIURE

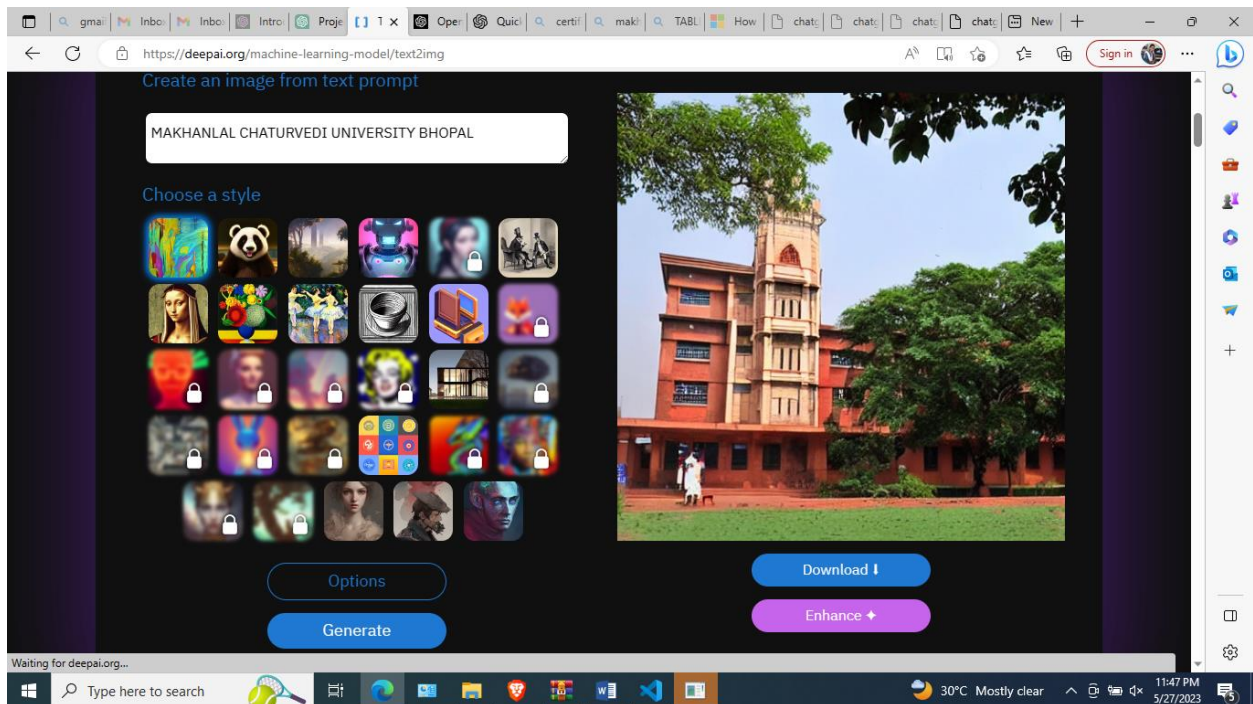
1.SPECIFIC UNIVERSITY PHOTO

2.VOICE CHAT

3.CHAT IN DATASTORE PROBLEM
 TAG

1.SPECIFIC UNIVERSITY PHOTO

I attempted to generate a specific photo of a university using the GPT-3 model for my Chat-GPT project, but unfortunately, I was unsuccessful. I searched through multiple photo websites, but I couldn't find a suitable photo of the university. The only site I found was Google Documentation, but it requires a visa, passport, and debit card for access. Since I don't possess a passport, visa, or debit card, I couldn't generate the necessary credentials and API key for obtaining a proper university photo.



2.VOICE CHAT

The functionality has been successful only in Python language, where the response is displayed on the print console. However, I have been unsuccessful in implementing voice chat on a web page. It might require Google credentials, but I haven't registered due to the unavailability of a visa, passport, and debit card.

3.CHAT IN DATASTORE PROBLEM
 TAG

I have successfully conducted a chat using the **GPT-3 model**. However, I encountered a problem where the "
" tag is being displayed in the chat store instead of creating a line break. I attempted to resolve this issue but was unsuccessful.

