# Implementación de funcionalidades básicas: Vistas y rutas
# © ADR Infor SL

# Indice

# Competencias y Resultados de Aprendizaje desarrollados en esta unidad

**Competencia:**
Diseñar, implementar y gestionar chatbots inteligentes e interactivos en aplicaciones web utilizando Django y Python, integrando habilidades de procesamiento de lenguaje natural para mejorar la experiencia del usuario y optimizar la interacción entre usuarios y chatbots.

**Resultados de Aprendizaje:**

- Implementar funciones básicas de procesamiento de texto usando las bibliotecas NLTK y spaCy.
- Realizar tokenización, análisis de partes del discurso y análisis de entidades nombradas para mejorar la comprensión del lenguaje por parte del chatbot.
- Crear vistas y rutas que permitan la interacción del usuario con el chatbot en una aplicación web Django.
- Configurar una base de datos PostgreSQL y gestionarla desde Django para almacenar las conversaciones del chatbot.
- Implementar la lógica del chatbot en views.py y vincularla con la interfaz de usuario a través de plantillas HTML y CSS.

# Implementación de funcionalidades básicas: Vistas y rutas

En esta unidad vamos a crear un chatbot limitado con funcionalidades básicas pero que tenga ya todo el aspecto y estructura para poder ir trabajando sobre él para crear nuestro chatbot.

## Creación del Proyecto y Configuración del Entorno Virtual

Aunque este punto debe estar claro como crear el proyecto para ponernos para ponernos a trabajar, vamos a realizar todos los pasos para la creación del proyecto y la configuración inicial de este para empezar a trabajar en el chatbot que queremos desarrollar.

## Instalación de Django y Configuración del Entorno Virtual con pipenv

Instalación de Django y configuración del entorno virtual con pipenv

Los pasos son los siguientes:

**Instalar pipenv si aún no está instalado**

```
pip install pipenv
```

**Crear un nuevo directorio y acceder a él**

También podemos hacerlo a través de la interface gráfica, si no desde cmd o desde consola, hacemos lo siguiente.

```
mkdir MyChatbot && cd MyChatbot
```

**Crear un entorno virtual y activarlo**

```
pipenv shell
```

**Instalar Django y todo lo que necesitamos para trabajar con él en desarrollo**

```
pipenv install django spacy numpy psycopg2 pillow
```

Es probable que falte algún plugin o que necesites realizar alguna configuración específica si vas a trabajar con Postgresql o quieres utilizar imágenes o archivos estáticos en desarrollo, para ello repasa los apartados anteriores.

# Creación del Proyecto Django y la app de chatbot

Creación del proyecto Django y app de chatbot

A continuación vamos a crear el proyeccto de Django como se supone que ya sabemos hacer, por si no lo hemos hecho ya

**Crear el proyecto:**

```
django-admin startproject MyChatbot .
```

**Crear una aplicación para el chatbot:**

```
python manage.py startapp chatbot
```

**Agregar chatbot en INSTALLED_APPS dentro de settings.py.**

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'chatbot',
]
```

# Configuración de la bbdd

De momento vamos a aparcar el proceso de creación de nuestro chatbot para centrarnos momentáneamente sobre la creación y configuración de la base de datos en postgresql.

Este paso es opcional pues si no quieres usar esta base de datos y quieres usar otra, puedes elegirlo a tu elección, de no hacer nada se creará una archivo sqlite3 que generará la base de datos.

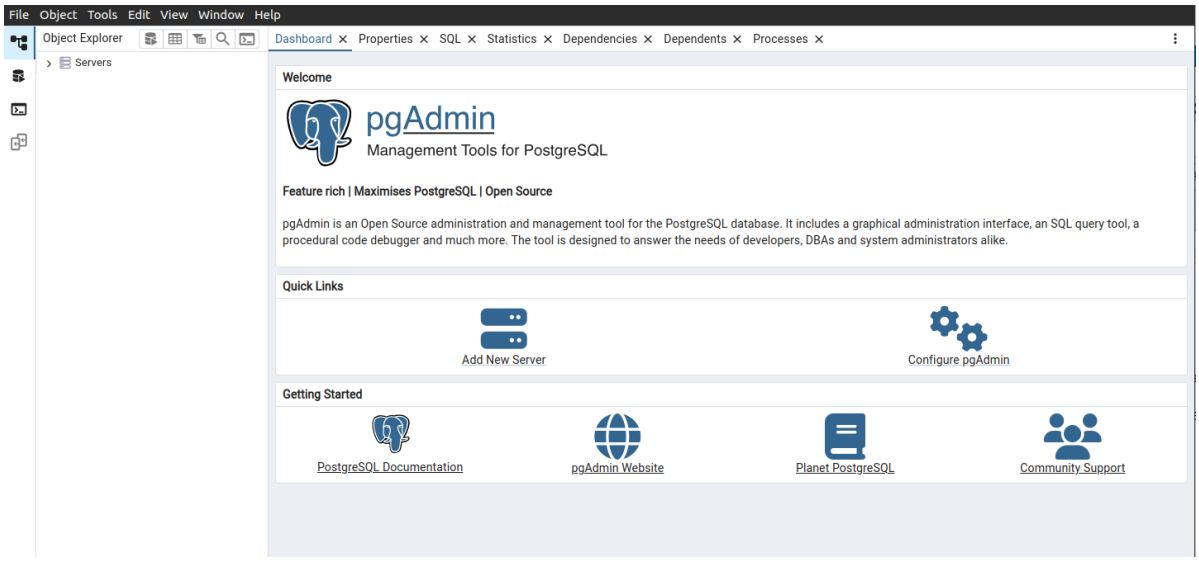
Para crear una base de datos en postgresql puedes hacerlo de dos formas, una es a través de pgadmin o a través de consola de comandos.

# Configuración POSTGRESQL a través de pgadmin

Vamos a ver los pasos para poder realizar este proceso:

**1**

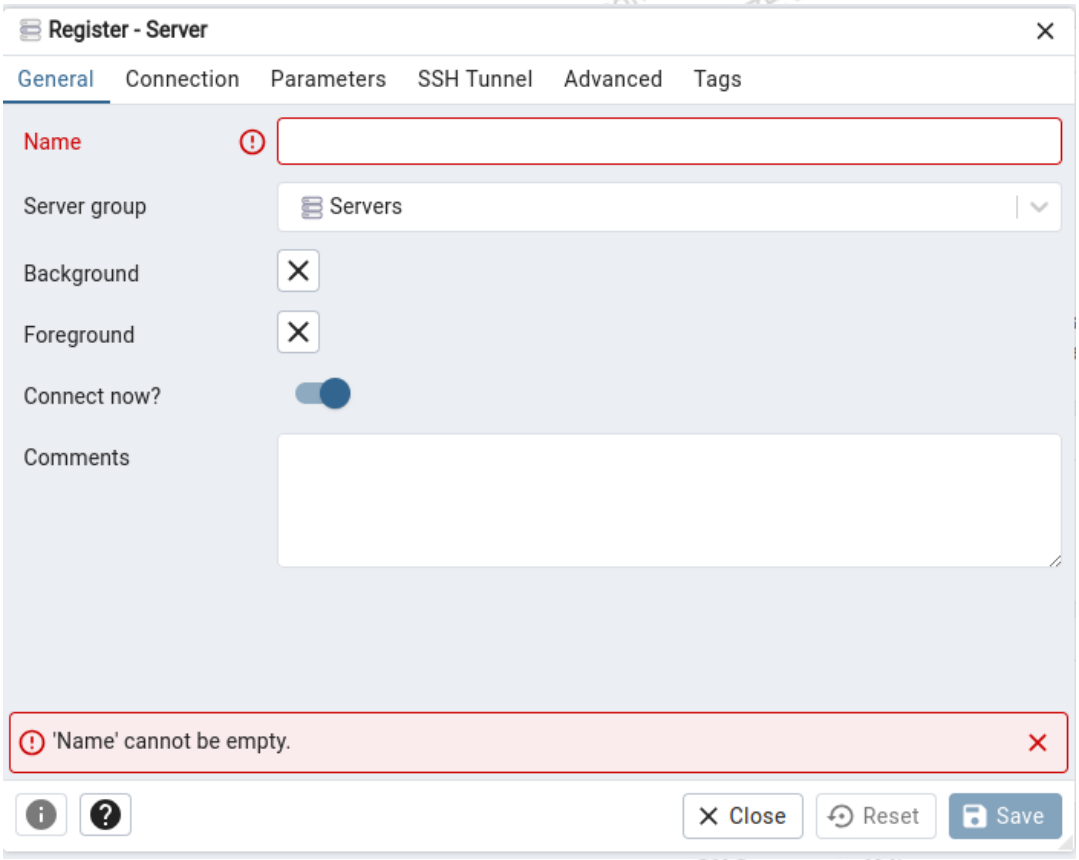
abrimos el programa pgadmin




**2**

creamos un servidor dentro de servers, para ello hacemos clic con el botón derecho register -> server

Una vez hecho esto nos aparecerá la siguiente ventana:




Rellenamos los siguientes elementos:

Dentro de General:

Name: Miservidor

Dentro de Connection:

host: localhost

port: 5432

Manteniance database: postgres

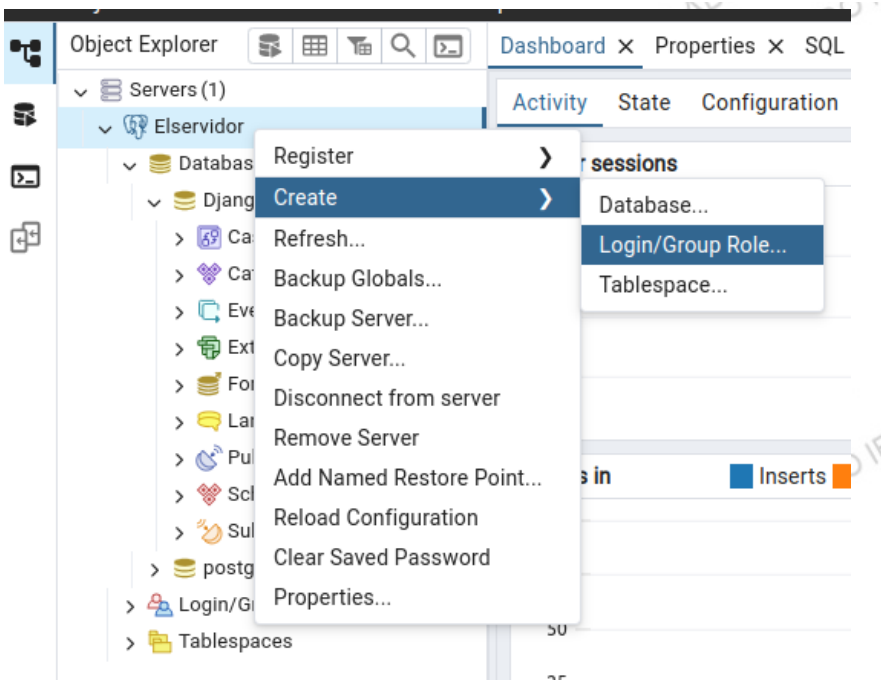
Username: postgres

Password: la contraseña que creaste al instalar postgresql

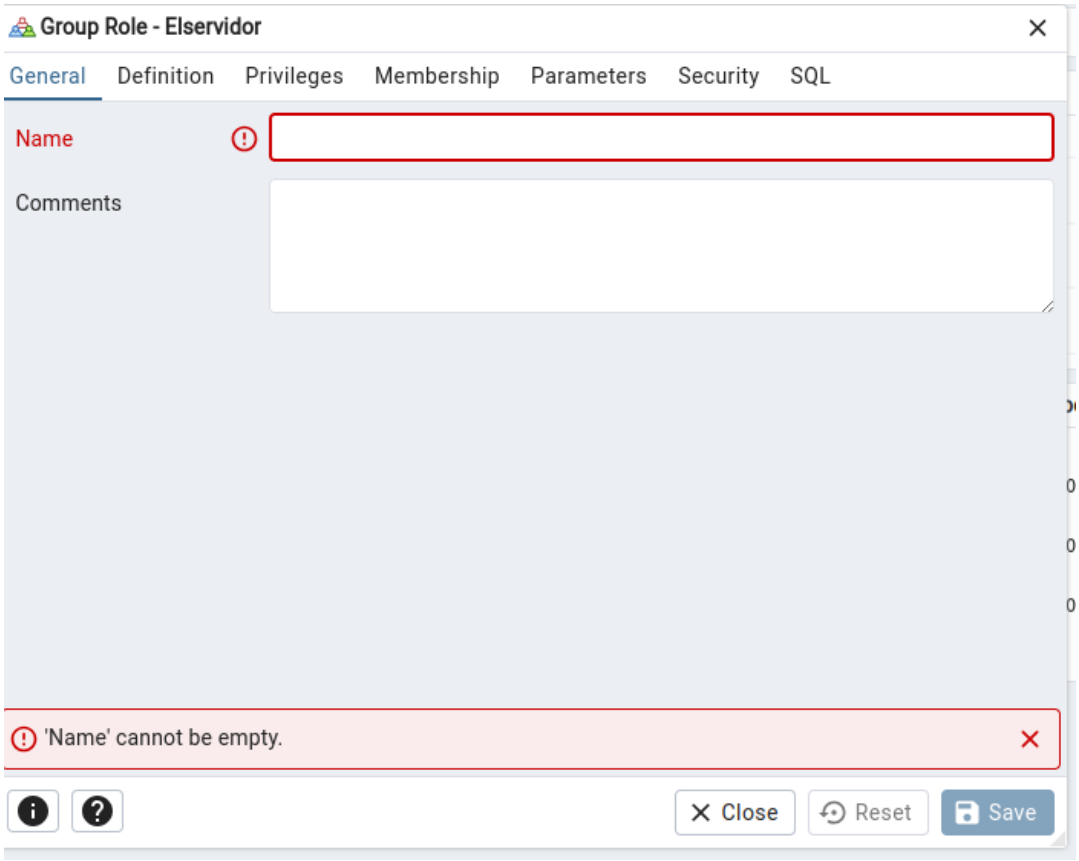
**3**

Ahora creamos un usuarios específico para este base de datos.

Para esto pulsamos con el botón derecho en el servidor y dentro del menú, seleccionamos




Y nos aparece la siguiente ventana:




Dentro de la pestaña General añadimos el Name

En Definition creamos la contraseña

En Privileges le damos privilegios de superuser

Damos a Guardar(save)

Ya tenemos creado nuestro usuario y nuestra base de datos por lo que ya podemos pasar a configurar el archivo settings.py de nuestro proyecto.

## Configuración POSTGRESQL a través consola

**Conectarte a PostgreSQL, abre CMD o PowerShell en windows o la terminal en linux**

**1**

Usa este comando para conectarte con el usuario postgres:(este es el superusuario por defecto de postgresql que se configura con su instalación)

Windows:

psql -U postgres

Linux:

sudo -u postgres psql

**2**

ya, dentro del postgres podemos crear el nuevo usuario que gestionará la base de datos, para crearlo teclearemos los siguiente comandos

-- Crear un usuario
CREATE ROLE mi_usuario WITH LOGIN PASSWORD 'mi_contraseña';

Con este comando, se crea un usuario llamado mi_usuario con la contraseña mi_contraseña, cambialos a tu elección.

-- Asignar privilegios (opcional, dependiendo de lo que necesites)
ALTER ROLE mi_usuario CREATEDB; -- Permite crear bases de datos
ALTER ROLE mi_usuario CREATEROLE; -- Permite crear otros roles

-- También puedes crear un superususario

ALTER ROLE mi_usuario SUPERUSER; -- Permite todos los privilegios

**3**

Una vez creado el usuario, puedes crear la base de datos usando este comando SQL:

```
-- Crear la base de datos
CREATE DATABASE mi_base_de_datos;

-- Otorgar permisos al usuario en la nueva base de datos
GRANT ALL PRIVILEGES ON DATABASE mi_base_de_datos TO mi_usuario;
```

Ya tenemos creado nuestro usuario y nuestra base de datos por lo que ya podemos pasar a configurar el archivo settings.py de nuestro proyecto.

## Configura la base de datos en settings.py:

En tu proyecto de Django, abre el archivo settings.py y busca la configuración de la base de datos. Cambia la configuración predeterminada de SQLite a PostgreSQL de la siguiente manera:

```
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.postgresql',
'NAME': 'nombre_de_tu_base_de_datos',
'USER': 'nombre_de_usuario',
'PASSWORD': 'tu_contraseña',
'HOST': 'localhost', # O la dirección de tu servidor PostgreSQL
'PORT': '5432', # El puerto por defecto de PostgreSQL
}
}
```

Asegúrate de reemplazar los valores de 'NAME', 'USER', 'PASSWORD', 'HOST', y 'PORT' con la configuración correcta de tu base de datos PostgreSQL.

# Creación de la Vista, la API y el template

Como ya sabemos, según nuestra arquitectura MVC, necesitamos crear la vista views.py que gestionará la información del usuario y como presentarla:

Creación de la vista, la API y el template

## Creación de views.py con la Lógica del Chatbot

No hemos creado una API propiamente dicha, pero sí hemos diseñado una vista que actúa como una API porque devuelve datos en formato JSON en lugar de renderizar una plantilla HTML.

### ¿Por qué la vista chatbot funciona como una API?

En Django, una vista normal (index) devuelve una página web (render(request, 'index.html')). En cambio, la vista chatbot responde con JSON, lo que permite que JavaScript la consulte sin necesidad de recargar la página.

## Diferencias clave entre una vista normal y una API en Django

| Característica | Vista Normal | API (como nuestra vista chatbot) |
|---|---|---|
| **Salida** | HTML (página web) | JSON (datos) |
| **Método de acceso** | El usuario la visita en el navegador | JavaScript la consulta con fetch() |
| **Recarga la página** | Sí | No, se actualiza dinámicamente |
| **Ejemplo de uso** | return render(request, 'index.html') | return JsonResponse({"response": "Hola"}) |

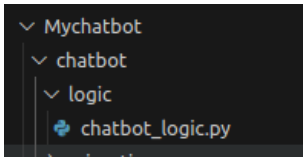
Después de todo este alegato, vamos a construir nuestra vista "API"

```
import json
from django.http import JsonResponse
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt
from .logic.chatbot_logic import get_response  # Importamos la lógica

def chatbot_view(request):
    return render(request, "chatbot/chatbot.html")

@csrf_exempt
def chatbot_api(request):
    if request.method == "POST":
        try:
            data = json.loads(request.body.decode("utf-8"))
            user_message = data.get("message", "").strip()
            if not user_message:
                return JsonResponse({"response": "No recibí un mensaje válido."}, status=400)
            bot_response = get_response(user_message)  # Llamamos a la lógica
            return JsonResponse({"response": bot_response})
        except json.JSONDecodeError:
            return JsonResponse({"response": "Error: JSON no válido."}, status=400)
    return JsonResponse({"response": "Método no permitido."}, status=405)
```

El decorador @csrf_exempt se utiliza solo en desarrollo, debes eliminarlo en la fase de producción, esto es muy importante para la seguridad del sitio ya que este decorador elimina la seguridad de los ataques csrf

## chatobt_logic.py

Para que esta "API" funcione correctamente vamos a crear un archivo dentro de la app chatbot que se llame chatbot_logic.py, la ruta de este archivo tiene que ser:




Este archivo va a ser el que controle la lógica de conversación del chatbot.

```
def get_response(user_message):
    """Devuelve una respuesta según el mensaje del usuario."""
    user_message = user_message.lower()

    responses = {
        "hola": "¡Hola! ¿Cómo puedo ayudarte?",
        "adiós":"¡Hasta luego! Que tengas un buen día.",
        "cómo estás":"Soy solo un programa, ¡pero gracias por preguntar!",
        "qué puedes hacer":"Puedo responder preguntas básicas. Inténtalo :)",
    }

    return responses.get(user_message, "No entiendo tu mensaje, ¿puedes reformularlo?")
```

## Funcionalidad "js" y estética "css" y plantilla "html" del chatbot

Continuando con el trabajo de visualización, para que se vea y funcione correctamente nuestro chatbot, vamos a crear la plantilla y los archivos necesarios para ello, supongo que sabes hacerlo pero por si acaso, realizo y explico todo el proceso.

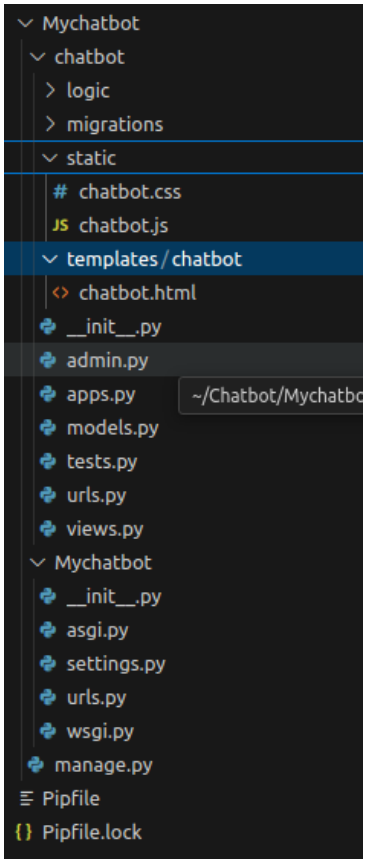
Organizaremos el código en tres archivos:

- **chatbot.html** (Estructura de la interfaz)
- **chatbot.css** (Estilos)
- **chatbot.js** (Lógica de interacción)

Lo primero que vamos a hacer es crear dos carpetas dentro de la app chatbot para la gestión de plantillas y archivos estáticos.

Para contener las plantillas, el camino más óptimo es crear una carpeta con este nombre dentro de la carpeta de cada aplicación, como en este caso solo tenemos la aplicación de cahtbot debemos crearla dentro de esta.

Para contener los archivos estáticos de la aplicación debemos optar por el mismo camino, es crear una carpeta con tal fin dentro de la aplicación. esta estructura de carpetas quedaría como sigue:






Donde la carpeta chatbot/static, la utilizaremos para los archivos estáticos, y la estructra chatbot/templates/chatbot/ la utilizaremos para almacenar los temas específicos de esta aplicación.

Una vez tenemos la configuración de las carpetas podemos empezar a crear los archivos que necesitamos.

**1**

Guárdalo en chatbot/templates/chatbot/chatbot.html

```
{% load static %}

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chatbot</title>
    <link rel="stylesheet" type="text/css" href="{% static 'chatbot.css' %}">
</head>
<body>

    <button id="chat-toggle">Chat</button>

    <div id="chat-container">
        <div id="chat-header">Chatbot</div>
        <div id="chat-messages"></div>
        <div id="chat-input-container">
            <input type="text" id="chat-input" placeholder="Escribe un mensaje...">
            <button id="send-button">Enviar</button>
        </div>
    </div>

    <script src="{% static 'chatbot.js' %}"></script>

</body>
</html>
```

**2**

A continuación creamos el archivo estático del estilo, esto siempre podrás modificarlo a tu gusto.

Guárdalo en static/chatbot.css.

```
#chat-container {

    position: fixed;

    bottom: 20px;

    right: 20px;

    width: 300px;

    height: 400px;

    background-color: white;

    border: 1px solid #ccc;

    border-radius: 10px;

    box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.2);

    display: none;
```

```
    flex-direction: column;
}

#chat-header {
    background-color: #007bff;
    color: white;
    padding: 10px;
    text-align: center;
    cursor: pointer;
}
#chat-messages {
    flex: 1;
    padding: 10px;
    overflow-y: auto;
}
#chat-input-container {
    display: flex;
    padding: 10px;
    border-top: 1px solid #ccc;
}

#chat-input {
    flex: 1;
    padding: 8px;
    border: 1px solid #ccc;
    border-radius: 5px;
} #send-button {
    background-color: #007bff;
    color: white;
```

```
    border: none;
    padding: 8px 12px;
    margin-left: 5px;
    border-radius: 5px;
    cursor: pointer;
} #send-button:hover {
    background-color: #0056b3;
} #chat-toggle {
    position: fixed;
    bottom: 20px;
    right: 20px;
    background-color: #007bff;
    color: white;
    border: none;
    width: 50px;
    height: 50px;
    border-radius: 50%;
    font-size: 20px;
    cursor: pointer;
    box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.2);
}
```

**3**

Guárdalo en static/chatbot.js.

```
document.addEventListener("DOMContentLoaded", function () {
    const chatToggle = document.getElementById("chat-toggle");
    const chatContainer = document.getElementById("chat-container");
    const chatMessages = document.getElementById("chat-messages");
    const chatInput = document.getElementById("chat-input");
```

```
 const sendButton = document.getElementById("send-button");
// Mostrar/ocultar chat
 chatToggle.addEventListener("click", function () {
    chatContainer.style.display = chatContainer.style.display === "none" ? "flex" : "none";
 });
// Enviar mensaje al hacer clic en el botón
 sendButton.addEventListener("click", sendMessage);
// Enviar mensaje al presionar Enter
 chatInput.addEventListener("keypress", function (event) {
    if (event.key === "Enter") {
        event.preventDefault();
        sendMessage();
    }
 });
function sendMessage() {
    const userMessage = chatInput.value.trim();
    if (userMessage === "") return;
   // Agregar mensaje del usuario al chat
    chatMessages.innerHTML += `<div><strong>Tú:</strong> ${userMessage}</div>`;
    chatInput.value = "";
   // Enviar al backend
    fetch("/chatbot/api/", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ message: userMessage }),
    })
    .then(response => response.json())
    .then(data => {
        chatMessages.innerHTML += `<div><strong>Bot:</strong> ${data.response}</div>`;
```

```
            chatMessages.scrollTop = chatMessages.scrollHeight;  // Auto-scroll
        })
        .catch(error => console.error("Error:", error));
    }
});
```

Debes asegurar que django encuentre los archivos estáticos, para ello debes modificar el archivo settings.py, ve al final del documento y añade las siguientes lineas.

```
import os

STATIC_URL = "/static/"
STATICFILES_DIRS = [os.path.join(BASE_DIR, "chatbot/static")]  # Asegura que Django encuentre los archiv
os estáticos
```

# Configuración de URL

Una muy buena costumbre es mantener las urls de cada aplicación dentro de estas, para ello lo que hecemos es **generar un archivo urls.py dentro de la aplicación** y después importarlo desde el archivo urls.py general del proyecto, así que:

En urls.py dentro de chatbot:

```
from django.urls import path
from .views import chatbot_api, chatbot_view

urlpatterns = [
    path('chatbot/', chatbot_view, name='chat_view'),
    path('chatbot/api/', chatbot_api, name='chat_api'),
]
```

En urls.py del proyecto, incluir:

```
from django.urls import include, path

urlpatterns = [
    path('', include('chatbot.urls')),
]
```

# Ejecución del Servidor

```
python manage.py runserver
```

Visitar http://127.0.0.1:8000/chatbot/ para probar el chatbot.

# Solución de posibles Problemas

Vamos a ver posibles problemas que nos vamos a encontrar durante la ejecución y desarrollo de nuestro código.

### Error no such table: chatbot_message

Causa: No se ejecutaron las migraciones. Solución:

python manage.py makemigrations
python manage.py migrate

### Error CSRF cookie not set

Causa: El servidor bloquea peticiones POST sin token CSRF. Solución: Se usa @csrf_exempt en la vista chat_api para pruebas locales.

### Error NameError: name 'json' is not defined

Causa: Falta importar la librería json. Solución: Agregar import json en views.py

## Resumen

- Inicia el proyecto configurando un entorno virtual con pipenv para gestionar dependencias. Posteriormente, crea el proyecto Django junto con la aplicación específica para el chatbot.
- Configura PostgreSQL utilizando pgAdmin o la consola. No olvides ajustar las configuraciones en el archivo settings.py para conectar correctamente la base de datos con Django.
- Desarrolla un modelo llamado Message que representará las interacciones del chatbot. Una vez definido, realiza y aplica las migraciones necesarias para reflejar los cambios en la base de datos.
- Crea una lógica básica para el chatbot en el archivo views.py. Esta lógica se encargará de procesar los mensajes enviados por los usuarios y generar las respuestas correspondientes.
- Configura las URLs necesarias para que la aplicación sea accesible desde el navegador. Utiliza el servidor de desarrollo de Django para validar que todo funcione correctamente.
- Prepara para solucionar posibles problemas que puedan surgir al ejecutar el servidor, como conflictos de puertos o errores de configuración, asegurándote de que la aplicación esté operativa.

# Actividades prácticas

## Integración de Chatbot con PostgreSQL en Django

En esta actividad, implementarás las configuraciones iniciales necesarias para integrar un chatbot en Django con una base de datos PostgreSQL. Sigue los pasos proporcionados y asegúrate de completar cada sección. Al final, tu entorno debe ser funcional y preparado para futuras implementaciones del chatbot.

1. Crea un entorno virtual e instala Django junto con las dependencias esenciales.
2. Inicia un nuevo proyecto Django llamado 'MyChatbot' e incluye una aplicación llamada 'chatbot'.
3. Configura PostgreSQL como el sistema de gestión de bases de datos para tu proyecto. Puedes usar pgAdmin o consola, según prefieras. Crea un usuario y una base de datos.
4. Actualiza el archivo settings.py de tu proyecto para reflejar la nueva configuración de PostgreSQL.

1. **Paso 1:** Crea y activa un entorno virtual, luego instala Django y las bibliotecas adicionales necesarias.

2. **Paso 2:** Crea un nuevo proyecto Django llamado 'MyChatbot' y añade una aplicación llamada 'chatbot'.

3. **Paso 3:** Crea un usuario y una base de datos en PostgreSQL. Puedes usar la interfaz gráfica de pgAdmin o la consola.

4. **Paso 4:** Configura settings.py para utilizar PostgreSQL.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos

## Integración de un Chatbot en Django

En esta actividad práctica, crearás un chatbot con Django y Python. El objetivo principal de esta práctica es diseñar, implementar y gestionar un chatbot funcional, configurar las vistas y rutas necesarias, y aplicar estilos básicos. Utilizarás el procesamiento de lenguaje natural básico para mejorar la experiencia del usuario. Se evaluará si eres capaz de integrar el chatbot como un componente interactivo dentro de una aplicación web de Django.

Pasos a seguir:

1. **Creación del Modelo:** Implementa el modelo Message dentro del archivo models.py, si aún no lo has hecho.
2. **Aplicación de Migraciones:** Aplica las migraciones necesarias para establecer el modelo en la base de datos.
3. **Implementación de la Vista API:** Dentro del archivo views.py, asegura que has creado la vista chatbot_api. Debe procesar peticiones POST y responder en formato JSON.
4. **Lógica del Chatbot:** En el archivo chatbot_logic.py, modifica la función get_response para incluir al menos 3 respuestas adicionales de tu elección, mejorando así las capacidades de respuesta del chatbot.
5. **Plantilla y Archivos Estáticos:** Configura correctamente los archivos chatbot.html, chatbot.css y chatbot.js para que el chatbot aparezca y funcione como se indica.✓ Plantilla HTML debe estar correctamente enlazada con sus archivos CSS y JS correspondientes.
6. **Configuración de Rutas:** Asegura que las rutas en urls.py del proyecto y de la aplicación estén correctamente configuradas para acceder al chatbot.



**1.** Comprueba que el modelo Message está correctamente implementado en models.py y las migraciones se han aplicado exitosamente.

**2.** Verifica si la vista chatbot_api responde adecuadamente a peticiones POST.

**3.** Añade al menos 3 respuestas más a la función get_response en chatbot_logic.py.

**4.** Asegura que los archivos HTML, CSS y JS se encuentran correctamente enlazados y el chatbot es visible en el navegador.

**5.** Cerciórate de que las rutas en urls.py están correctamente configuradas para el funcionamiento del chatbot.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos