Integración de Modelos en el Chatbot
(De Memory a Modelos Django)
© 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:

- Gestionar el contexto de la conversación para mantener la coherencia entre los mensajes del chatbot y las respuestas del usuario.
- Integrar modelos de Django en el flujo conversacional del chatbot para almacenar y recuperar información de las interacciones.
- Configurar un sistema de logs en el chatbot para registrar las interacciones y errores.

Integración de Modelos en el Chatbot (De Memory a Modelos Django)

Hasta ahora hemos estado trabajando con memoria volátil para almacenar la información del chatbot. Pero eso no es escalable ni fiable.
¿Qué pasa si el servidor se reinicia? Perdemos toda la conversación.

Por eso, en esta unidad vamos a integrar los modelos de Django en nuestro flujo conversacional.
¿La buena noticia? Ya sabes crear modelos. Lo viste en detalle en la Unidad 6.
Ahora el foco es otro: aprender a usarlos dentro del chatbot para guardar usuarios, conversaciones, mensajes y estados.

Estructura de los Modelos Conversacionales

En esta parte no vamos a inventar la rueda.
Vamos a aplicar lo que ya sabemos sobre modelos en Django, retomando con mirada sobre la del diseño conversacional.

El objetivo no es solo guardar datos porque "hay que guardarlos", sino estructurar **la experiencia conversacional** como una app real.

El Modelo que vamos a usar

Aquí está el reparto protagonista en este caso con un único actor de nuestro chatbot:

Reserva (El modelo)

En el momento en que se termina el flujo de la conversación y se confirma la reserva, esta se guarda el bbdd a través del modelo.

Este modelo y logger bastan para levantar un chatbot que no solo responda, sino que **recuerde**, **organice** y **entienda el contexto**.

Pero, ¿Qué es logger en Python?

Un logger es simplemente un sistema para escribir mensajes en un registro (log).

NO guarda datos del navegador como las cookies. NO es algo que "viaje por internet".

Es interno del servidor o de la app. Para ti, como desarrollador.

En definitva, usamos este método por que nos ayuda a **registrar eventos** en el **servidor**:

- Guardar qué mensajes entran, para cambiar de estado en la conversación.
- Anotar errores si algo falla, para depurar.
- Dejar constancia de qué flujo siguió un usuario en el chatbot,

- Ayudarte a **debuguear** si algo no funciona.

Diferencias fundamentales entre cookies y logger

La app es un chatbot de reservas que vive en el servidor.
Se controla todo desde el backend (servidor). No estamos haciendo una web con navegador, sino un sistema que procesa mensajes (y luego quizás responde en una web, WhatsApp, etc.).

¿Para qué nos sirve esta estructura?

Podríamos guardar los mensajes como una lista en memoria, claro. Pero si lo hacemos bien desde el principio, conseguimos mucho más:

- Llevar un **historial real** de cada conversación.
- **Personalizar respuestas** según el usuario (incluso si vuelve días después).
- **Retomar conversaciones interrumpidas** sin que el bot se quede en blanco.
- Permitir análisis posteriores: tasa de reservas, tasa de repetición de los clientes, etc.

Y lo más importante: **estamos sentando las bases para un chatbot que escale y evolucione** .

Achivo model.py

```
 # models.py
from django.db import models

class Reserva(models.Model):
    nombre = models.CharField(max_length=100)
    fecha = models.DateTimeField()
    numero_personas = models.IntegerField()
    alergias = models.TextField(blank=True, null=True)
    telefono = models.CharField(max_length=20)
    creado_en = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Reserva de {self.nombre} para {self.numero_personas} personas el {self.fecha.strftime('%d/%m/
%Y %H:%M')}"
```

Integración del Flujo Conversacional en los Modelos

Ahora que ya tenemos clara la estructura del modelo, toca lo importante: **ponerlo en acción**.

¿Qué significa integrar el flujo conversacional?

Significa que cada paso que da el chatbot deja un **rastro estructurado** en las cookies que utilizamos para poder mantener el flujo. Cuando se completa la reserva, esta queda registrado.
Esto no solo nos da orden: **nos da memoria, contexto y control**.

El usuario comienza una conversación, el bot le ofrece las diferentes etapas por las que llevar la reserva, para ir almacenando cada etapa se utiliza un estado temporal que nos sirve para avanzar.

Nuestros archivos

- chatbot_logic.py → estructura conversacional, gestión de estados, lógica condicional.
- nlp_utils.py → análisis de texto, detección de intención, preprocesamiento con spaCy, dateparser, etc.

Chatbot_logic.py

El código

```
 import logging
from datetime import datetime
from ..models import Reserva
import dateparser
from .nlp_utils import detectar_intencion, extraer_fecha, es_afirmacion, extraer_personas, limpiar_y_validar_tel
efono, normalizar_mensaje
import re

# Configura el logger
logger = logging.getLogger("chatbot")
logger.setLevel(logging.DEBUG)

# Estados como constantes
STATE_SALUDO = 'saludo'
STATE_VER_MENU = 'ver_menu'
STATE_RESERVA = 'reserva'
STATE_NOMBRE = 'nombre'
STATE_FECHA = 'fecha'
STATE_PERSONAS = 'personas'
STATE_ALERGIAS = 'alergias'
STATE_TELEFONO = 'telefono'
STATE_CONFIRMACION = 'confirmacion'
STATE_DESPEDIDA = 'despedida'

def get_response(user_message, session_id, session_memory):
    try:
        message_content = normalizar_mensaje(user_message, session_id, logger)
        logger.debug(f"[{session_id}] Contenido normalizado: {message_content} | Tipo: {type(message_content)}
")

        state = session_memory.get('state', STATE_SALUDO)
        #logger.debug(f"[{session_id}] Estado actual: {state} | Mensaje recibido: {user_message}")
```

```
    logger.debug(f"[{session_id}] Estado actual: {state} | Input: {message_content} | Memoria: {session_mem
ory}")
    responses = {
        STATE_SALUDO: "¡Bienvenido! ¿Deseas ver nuestro menú o prefieres hacer una reserva?",
        STATE_VER_MENU: "Aquí tienes nuestro menú. ¿Quieres reservar una mesa ahora?",
        STATE_RESERVA: "Perfecto, comencemos con tu reserva. ¿Cómo te llamas?",
        STATE_FECHA: "Gracias {nombre}. ¿Para qué día te gustaría reservar? (Por ejemplo: 'mañana', '15 de
 julio')",
        STATE_PERSONAS: "Entendido. ¿Para cuántas personas será la reserva? (Máximo 25 personas)",
        STATE_ALERGIAS: "¿Algún comensal tiene alergias alimentarias o necesidades especiales que deba
mos conocer?",
        STATE_TELEFONO: "Necesitamos un número de contacto para confirmar tu reserva (9 dígitos, por ej
emplo: 612345678)",
        STATE_CONFIRMACION: "  Reserva confirmada! ¿Necesitas algo más?",
        STATE_DESPEDIDA: "¡Gracias por elegirnos! Esperamos verte pronto. ¡Buen día!"
}
    if state == STATE_SALUDO:
        if 'menu' in message_content.lower():
            session_memory['state'] = STATE_VER_MENU
            return responses[STATE_VER_MENU]
        elif 'reserva' in message_content.lower() or es_afirmacion(message_content):
            session_memory['state'] = STATE_RESERVA
            return responses[STATE_RESERVA]
        return responses[STATE_SALUDO]

    elif state == STATE_VER_MENU:
        if 'reserva' in message_content.lower() or es_afirmacion(message_content):
            session_memory['state'] = STATE_RESERVA
            print(f"[{session_id}] PAsa a Reserva??: {message_content}")
            return responses[STATE_RESERVA]

        return responses[STATE_VER_MENU]

    elif state == STATE_RESERVA:
        print(f"[{session_id}] Estado user_message: {message_content}")
        session_memory['nombre'] = message_content
        nombre= session_memory['nombre']
        session_memory['state'] = STATE_FECHA
        return f"Gracias {nombre}. ¿Para qué día te gustaría reservar? (Por ejemplo: 'mañana', '15 de julio')"#r
esponses[STATE_FECHA]


    elif state == STATE_FECHA:

        valido, fecha = extraer_fecha(message_content)
        if not valido:
            logger.warning(f"[{session_id}] Fecha inválida: {message_content}")
            return "No entendí la fecha. Por favor usa un formato como '10 de mayo' o 'próximo lunes'"

        #session_memory['fecha'] = fecha
        session_memory['fecha'] = fecha.isoformat()  # o .strftime("%Y-%m-%d")
        print(f"[{session_id}] Fecha válida: {message_content}")
        session_memory['state'] = STATE_PERSONAS  # Pasar a personas después de fecha válida
        return responses[STATE_PERSONAS]  # "Perfecto, ¿Cuántas personas asistirán?"

    elif state == STATE_PERSONAS:  # Nuevo bloque para manejar número de personas
        valido, personas = extraer_personas(message_content)
        if not valido:
```



```
            return "Por favor indica un número entre 1 y 25 personas"

        session_memory['personas'] = personas
        session_memory['state'] = STATE_ALERGIAS
        return responses[STATE_ALERGIAS]

    elif state == STATE_ALERGIAS:
        session_memory['alergias'] = message_content
        session_memory['state'] = STATE_TELEFONO
        return responses[STATE_TELEFONO]

    elif state == STATE_TELEFONO:
        valido, telefono = limpiar_y_validar_telefono(message_content)
        if valido:
            session_memory['telefono'] = telefono
            session_memory['state'] = STATE_CONFIRMACION
            return responses[STATE_CONFIRMACION]
        else:
            return "Por favor, proporciona un número de teléfono válido."

    elif state == STATE_CONFIRMACION:
        nombre = session_memory.get('nombre', 'Anónimo')
        #fecha = session_memory.get('fecha')
        numero_personas = session_memory.get('personas')
        alergias = session_memory.get('alergias', '')
        telefono = session_memory.get('telefono')

        #fecha_str = fecha.strftime("%d/%m/%Y %H:%M") if fecha else "Fecha no proporcionada"
        fecha_str = session_memory.get('fecha')
        fecha_dt = datetime.fromisoformat(fecha_str) if fecha_str else None
        fecha_str = fecha_dt.strftime("%d/%m/%Y %H:%M") if fecha_dt else "Fecha no proporcionada"

        # Crear la reserva en la base de datos
        Reserva.objects.create(
            nombre=nombre,
            fecha=fecha_dt,
            numero_personas=numero_personas,
            alergias=alergias,
            telefono=telefono,
        )

        logger.info(f"[{session_id}] Reserva confirmada para {numero_personas} personas.")

        session_memory.clear()
        session_memory['state'] = STATE_SALUDO
        return f"¡Gracias! Tu reserva para {numero_personas} personas ha sido confirmada para el {fecha_str}.
"

    elif state == STATE_DESPEDIDA:
        return responses[STATE_DESPEDIDA]

    return "Lo siento, no entiendo esa solicitud."

  except Exception as e:
    logger.exception(f"[{session_id}] Error en el procesamiento del mensaje: {e}")
    session_memory['state'] = STATE_SALUDO
    return "Algo salió mal procesando tu mensaje. ¿Podrías intentarlo de nuevo?"
```

Cómo funciona el código

Lo primero, las importaciones

- **logging:** para registrar todo (errores, info, etc).
- **datetime**: para trabajar con fechas y horas.
- **Reserva:** el modelo de base de datos donde guardamos las reservas.
- **dateparser:** para interpretar fechas escritas de forma humana ("mañana", "el 10 de mayo"...).
- **nlp_utils**: funciones inteligentes para detectar intenciones, extraer fechas y números de personas, limpiar teléfonos, normalizar mensajes.

Configuración del logger

un logger es un objeto que se utiliza para registrar eventos durante la ejecución de un programa. Permite registrar mensajes, errores y otros detalles de la ejecución, que luego pueden ser útiles para depurar, monitorear o analizar el comportamiento del programa.

- Crea un sistema de registro para ver qué está pasando por dentro: mensajes, errores, cambios de estado.

- Nivel DEBUG: **muestra absolutamente todo**

Estados de la conversación

```
STATE_SALUDO = 'saludo'
STATE_VER_MENU = 'ver_menu'
STATE_RESERVA = 'reserva'
...
```

- Cada interacción con el usuario depende de en qué estado esté la conversación.
- El chatbot no improvisa: siempre está en un estado, y sabe qué esperar a continuación.

Función get_response(user_message, session_id, session_memory)

Cada vez que el usuario manda un mensaje:

- user_message: el mensaje de texto.
- session_id: el ID de la conversación.
- session_memory: un diccionario que guarda todo lo que el usuario ya dijo (nombre, fecha, etc.).

Flujo conversación según el estado

STATE_SALUDO

- Se oferta al usuario los caminos posibles por los que llevar la conversación.
- Si el usuario dice "menú" → muestra menú.
- Si dice "reservar" o una afirmación ("sí", "ok") → empieza la reserva.
- Si no, sigue preguntando como loro.

STATE_VER_MENU

- Igual: si tras ver el menú el usuario quiere reservar, pasa a reservar.

STATE_RESERVA

- Guarda el nombre del usuario y pregunta cuándo quiere reservar.

STATE_FECHA

- Usa extraer_fecha() para entender la fecha.
- Si no entiende, pide que escriba de nuevo.
- Si la fecha es válida, sigue preguntando para cuántas personas.

STATE_PERSONAS

- Usa extraer_personas() para entender el número de personas (máx. 25).
- Si no es válido, pide repetir.

STATE_ALERGIAS

- Guarda si alguien tiene alergias o necesidades especiales.

STATE_TELEFONO

- Usa limpiar_y_validar_telefono() para pedir y validar un número de contacto.

STATE_CONFIRMACION

- Guarda toda la información en la base de datos:

```
Reserva.objects.create(
nombre=nombre,
fecha=fecha_dt,
numero_personas=numero_personas,
alergias=alergias,
telefono=telefono,
)
```

- Limpia la session_memory para empezar otra reserva si hace falta.
- Confirma la reserva al usuario.

Manejo de errores... Si algo sale mal

```
except Exception as e:
logger.exception(f"[{session_id}] Error en el procesamiento del mensaje: {e}")
session_memory['state'] = STATE_SALUDO
return "Algo salió mal procesando tu mensaje. ¿Podrías intentarlo de nuevo?"
```

- No explota el chatbot.
- Devuelve un mensaje genérico y vuelve al estado inicial.

nlp_utils.py

El código

```
 # Importación de librerías necesarias:

import spacy  # Librería para el procesamiento del lenguaje natural
import re  # Librería para expresiones regulares, usada para limpiar el texto
import dateparser  # Librería para el análisis y parseo de fechas en texto
from dateparser.search import search_dates
from datetime import datetime,timedelta
import difflib
import logging
from logging import Logger

# Cargamos el modelo de lenguaje en español de spaCy
nlp = spacy.load("es_core_news_md")

# Definimos una lista de posibles saludos que el bot puede reconocer
SALUDOS = ["hola", "buenos días", "buenas tardes", "qué tal", "hey", "saludos"]

# Definimos una lista de posibles afirmaciones que el bot puede reconocer
AFIRMACIONES = ["si", "claro", "vale", "de acuerdo", "por supuesto", "si quiero", "ok"]
NUMEROS_PALABRAS = {
    "uno": 1, "una": 1, "dos": 2, "tres": 3, "cuatro": 4, "cinco": 5,
    "seis": 6, "siete": 7, "ocho": 8, "nueve": 9, "diez": 10,
    "once": 11, "doce": 12, "trece": 13, "catorce": 14, "quince": 15,
    "dieciséis": 16, "diecisiete": 17, "dieciocho": 18, "diecinueve": 19,
    "veinte": 20, "veintiuno": 21, "veintidós": 22, "veintitrés": 23,
    "veinticuatro": 24, "veinticinco": 25,
    "pareja": 2, "una pareja": 2
}

# Función para detectar la intención del usuario en base al texto introducido

def detectar_intencion(texto):
    """
    Esta función detecta si el texto introducido por el usuario corresponde a un saludo.
    Si se detecta un saludo, devuelve 'saludo'; de lo contrario, devuelve 'desconocida'.
    """
    texto = texto.lower()  # Convertimos el texto a minúsculas para normalizar la comparación
    for saludo in SALUDOS:
        if saludo in texto:  # Si uno de los saludos está presente en el texto
            return "saludo"  # Retornamos 'saludo' si se detecta un saludo
    return "desconocida"  # Si no se detecta un saludo, retornamos 'desconocida'

# Función para determinar si el texto del usuario contiene una afirmación

def es_afirmacion(texto):
    """
    Esta función verifica si el texto del usuario contiene alguna afirmación,
    permitiendo variaciones y utilizando similitud de texto para reconocer afirmaciones
    incluso si el usuario escribe algo como "claro que sí" o "sí por supuesto".
    """
    texto = texto.strip().lower()  # Normalizamos el texto a minúsculas

    # Normalizamos 'sí' a 'si'
    texto = texto.replace("sí", "si")
```

```
    # Eliminar palabras repetidas y limpiar espacios extra
    texto = " ".join(sorted(set(texto.split()), key=texto.split().index))  # Eliminar repeticiones

    # 1. Coincidencia exacta directa
    if texto in AFIRMACIONES:
        return True

    # 2. Contiene alguna palabra clave
    for afirm in AFIRMACIONES:
        if afirm in texto:
            return True

    # 3. Similitud con alguna afirmación (por si acaso)
    for afirm in AFIRMACIONES:
        similitud = difflib.SequenceMatcher(None, texto, afirm).ratio()
        if similitud > 0.8:
            return True

    return False

# Función para extraer una fecha de un texto dado

def limpiar_input_fecha(texto):
    texto = texto.lower().strip()

    # Detecta "el jueves que viene" y cambia por "próximo jueves"
    match = re.match(r"(el\s+)?(?P<dia>\w+)\s+que\s+viene", texto)
    if match:
        dia = match.group("dia")
        texto = f"próximo {dia}"

    # "este jueves" → "jueves"
    texto = re.sub(r"\beste\b\s+(lunes|martes|miércoles|jueves|viernes|sábado|domingo)\b", r"", texto)

    # Casos simples: "el próximo lunes" → "próximo lunes"
    texto = re.sub(r"\bel\b", "", texto)
    texto = re.sub(r"\s+", " ", texto).strip()

    return texto

def extraer_fecha(user_input):
    limpio = limpiar_input_fecha(user_input)
    print(f"[DEBUG] Fecha interpretada: '{limpio}'")

    # Usamos search_dates para detectar fechas relativas
    resultados = search_dates(
        limpio,
        languages=["es"],
        settings={
            "PREFER_DATES_FROM": "future",
            "RELATIVE_BASE": datetime.now()
        }
    )

    if not resultados:
        return False, None

    # Tomamos la primera fecha que encuentre
```

```
    _, fecha = resultados[0]  # Obviamos el primer valor (que es el texto original)

    # Validación de la fecha para asegurar que sea razonable
    hoy = datetime.now()

    if fecha.year < hoy.year - 1 or fecha.year > hoy.year + 2:
        return False, None

    return True, fecha  # Devuelvo la fecha sin formatear

def formatear_fecha_es(fecha):
    # Diccionarios para los días y meses en español
    dias = {
        "Monday": "lunes", "Tuesday": "martes", "Wednesday": "miércoles",
        "Thursday": "jueves", "Friday": "viernes", "Saturday": "sábado", "Sunday": "domingo"
    }
    meses = {
        "January": "enero", "February": "febrero", "March": "marzo",
        "April": "abril", "May": "mayo", "June": "junio", "July": "julio",
        "August": "agosto", "September": "septiembre", "October": "octubre",
        "November": "noviembre", "December": "diciembre"
    }

    # Convertir la fecha al formato adecuado en español
    dia_en = fecha.strftime("%A")
    mes_en = fecha.strftime("%B")
    dia = dias[dia_en]
    mes = meses[mes_en]

    return f"{dia} {fecha.day} de {mes}"


def extraer_personas(texto):
    texto = texto.lower().strip()

    # Primero intentamos encontrar un número escrito con dígitos
    match_num = re.search(r"\b(\d{1,3})\b", texto)
    if match_num:
        num = int(match_num.group(1))
        if 1 <= num <= 25:
            return True, num
        else:
            return False, None  # Fuera de rango

    # Si no hay dígitos, buscamos palabras
    for palabra, valor in NUMEROS_PALABRAS.items():
        if palabra in texto:
            if 1 <= valor <= 25:
                return True, valor
            else:
                return False, None  # También fuera de rango

    # Nada útil encontrado
    return False, None
import re

def limpiar_y_validar_telefono(user_input):
    """
```

```
    Limpia el input y valida si es un número de teléfono español válido (9 dígitos).
    Devuelve (True, telefono_limpio) si es válido, o (False, None) si no.
    """
    if not user_input:
        return False, None

    # Convertimos a string, eliminamos todo excepto dígitos
    telefono_limpio = re.sub(r"\D", "", str(user_input))

    # Validamos que tenga exactamente 9 dígitos
    if len(telefono_limpio) == 9:
        return True, telefono_limpio
    return False, None
```

```
import json

def normalizar_mensaje(raw_message, session_id=None, logger=None):
    """
    Convierte un mensaje entrante en un string plano.
    Si es un dict tipo {"message": "texto"}, extrae el valor.
    Si es un JSON stringificado, lo parsea.
    """
    try:
        if isinstance(raw_message, str) and raw_message.strip().startswith("{"):
            # Intenta decodificar si parece JSON
            raw_message = json.loads(raw_message)
            if logger and session_id:
                logger.debug(f"[{session_id}] Mensaje JSON decodificado correctamente.")
    except Exception as e:
        if logger and session_id:
            logger.warning(f"[{session_id}] No se pudo parsear el JSON: {e}")
        pass

    if isinstance(raw_message, dict):
        return str(raw_message.get("message", "")).strip()
    return str(raw_message).strip()
```

Cómo funciona el código

Importaciones

Carga librerías de NLP (spacy, dateparser), expresiones regulares (re), manejo de fechas (datetime), comparación de textos (difflib) y logging (logging). Todo para que el bot pueda leer, limpiar, interpretar y loguear lo que recibe.

Carga del modelo de lenguaje

```
nlp = spacy.load("es_core_news_md")
```

Carga un modelo de spaCy en español para entender mejor el lenguaje humano. No lo ves usado mucho aquí directamente, pero queda cargado para funciones más avanzadas si se necesitan.

Diccionarios y listas base

SALUDOS: palabras que identifican si el usuario saluda.

AFIRMACIONES: palabras que detectan si el usuario dice que "sí".

NUMEROS_PALABRAS: mapeo de palabras como "tres", "pareja", etc., a números.

Funciones utilitarias

detectar_intencion(texto)

- Mira si el usuario saludó usando las palabras de SALUDOS.
- Devuelve "saludo" o "desconocida".
- Básicamente un detector de "¿Me está diciendo hola o no?".

es_afirmacion(texto)

Detecta si el usuario está diciendo algo afirmativo ("sí", "claro", "ok", etc.).

Usa 3 niveles de inteligencia:

- Coincidencia exacta.
- Contiene una afirmación como substring.
- Similitud de texto (por si escriben algo parecido, pero no exacto).

extraer_fecha(user_input)

- Saca una fecha del texto, incluso si dicen "el jueves que viene" o "mañana".
- Usa dateparser para entender español.
- Se asegura de que la fecha tenga sentido (ni 1950 ni 2080).

limpiar_input_fecha(texto)

- Preprocesa el texto para que dateparser no se vuelva loco.
- Ejemplo: "el jueves que viene" → "próximo jueves".

formatear_fecha_es(fecha)

Cambia una fecha de Python (datetime) a texto bonito en español: "lunes 15 de julio", no "Monday, July 15".

extraer_personas(texto)<

- Encuentra cuántas personas dicen que irán.
- Detecta tanto números ("5") como palabras ("cinco", "una pareja").

limpiar_y_validar_telefono(user_input)

- Limpia cualquier cosa rara (espacios, guiones, letras) y se queda solo con los números.
- Se asegura de que haya exactamente 9 dígitos (formato de teléfono español).

normalizar_mensaje(raw_message, session_id=None, logger=None)

- Convierte cualquier input en un string plano.

- Si recibe un JSON tipo {"message": "hola"}, saca el "hola".
- Además deja todo prolijo para que el bot no explote cuando reciba basura.

Otros archivos importantes

Archivos que ya hemos trabajado como ***chatbot.js, chatbot.css, settings.py y chatbot.html*** son los mismos con los que ya habiamos trabajado, no tienes porque modificarlos

La administración y visualización de la reserva

Solo nos queda ver cómo accedemos a los mensajes.

Crear el superusuario

Para poder acceder a la zona de administración de Django, necesitamos un usuario con permisos suficiente.:

1. Abre un terminal o consola.
2. Asegúrate de estar en el directorio donde se encuentra tu proyecto Django. Si no estás en esa carpeta, usa el comando cd para navegar hasta ella.
3. Una vez que estés en el directorio del proyecto, ejecuta el siguiente comando para crear un superusuario:
    - python manage.py createsuperuser
4. Django te pedirá que ingreses algunos datos para crear el superusuario. Debes proporcionar:
    - **Nombre de usuario**: Este será el nombre con el que iniciarás sesión en el panel de administración de Django.
    - **Dirección de correo electrónico**: El correo electrónico del superusuario.
    - **Contraseña**: Se te pedirá que introduzcas una contraseña, y luego te pedirá que la confirmes para asegurarse de que la hayas escrito correctamente.
5. Si el servidor no está en ejecución, puedes arrancarlo como siempre.
6. Ahora podrás acceder al panel de administración de Django. Abre tu navegador y ve a la siguiente URL:
    - http://127.0.0.1:8000/admin/

Código admin.py

La primera versión en admin.py incluye el modelo de reservas en el panel de administración con algunos campos básicos como nombre, fecha, numero_personas y telefono, y la capacidad de realizar búsquedas sobre los campos nombre y fecha. Esto ya es un buen punto de partida para la gestión de reservas.

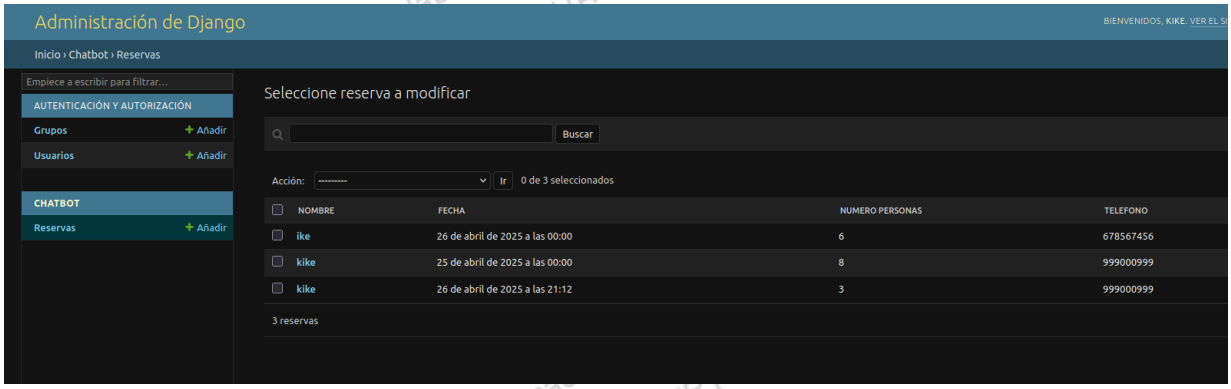
```
 from django.contrib import admin
from django.utils.html import format_html
from .models import Reserva

# Registro del modelo en la administración
# @admin.register(Reserva)  # Comentado, ya que no se usa este decorador directamente
class ReservaAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'fecha', 'numero_personas', 'telefono', 'ver_detalles')
    search_fields = ('nombre', 'fecha')

# Registrar el modelo
admin.site.register(Reserva, ReservaAdmin)
```

Con esto, las reservas se presentaban en una lista básica en el panel de administración y ya tenías la funcionalidad de búsqueda.

Veremos algo así



Añadiendo lógica de permisos de edición condicional

Una mejora importante es añadir un control sobre si el administrador puede editar una reserva o no. Esto se conseguimos implementando el método has_change_permission para verificar si el parámetro edit=True está presente en la URL. Si no se encuentra, el administrador no podrá editar esa reserva.

Hacemos para evitar modificaciones imprevistas y que estas se realicen de forma totalmente conscientes.

Añandimos a continuación dentro de la clase ReservaAdmin, el siguiente código:

```
def has_change_permission(self, request, obj=None):
    # Si el usuario no ha solicitado editar, no permite cambios
    if obj is not None and not request.GET.get('edit', False):  # Lógica de bloqueo de edición
        return False
    return super().has_change_permission(request, obj)
```

- **has_change_permission** se usa para controlar si el usuario tiene permiso para cambiar el objeto. En tu caso, se añaden condiciones para que el administrador **no pueda editar** si no se pasa el parámetro edit=True en la URL.

- **Lógica**: Cuando el administrador intenta ver una reserva, si no tiene el parámetro edit=True en la URL, la función devolverá False, lo que significa que no se podrá editar esa reserva.

- **Esto asegura** que solo cuando el administrador haga clic en un enlace con edit=True podrá editar la reserva.

Añadiendo el botón de "Ver detalles" con enlace a la edición

Para hacerlo aún más intuitivo y permitir a los administradores editar reservas solo cuando lo deseen, añades un botón en la lista de reservas que les lleva a la página de edición. Este botón está habilitado solo si el administrador hace clic explícitamente en el enlace.

Código final con botón "Ver detalles" y enlace a edición:

```
def ver_detalles(self, obj):
    # Crear un enlace para la edición de la reserva
    return format_html(
        ''
        '<a href="/admin/chatbot/reserva/{}/change/">✏ Editar</a>',
        obj.id
    )
ver_detalles.short_description = 'Detalles'
```

- **ver_detalles**: Este método crea un enlace HTML con el texto **"✏ Editar"** que lleva al administrador a la página de edición de la reserva, pero sólo si hace clic en ese enlace. La URL incluye ?edit=True, lo que activa el permiso para editar.
    - format_html se usa para evitar problemas de inyección de HTML.
    - href="/admin/chatbot/reserva/{}/change/?edit=True" genera un enlace que contiene el parámetro edit=True, permitiendo la edición de la reserva.
- **ver_detalles.short_description = 'Detalles'**: Cambia el texto de la columna que aparece en la lista para que sea más claro, en lugar de un nombre genérico como "ver".

El cñodigo completo de la clase ReservaAdmin es:

```
 from django.contrib import admin
from django.utils.html import format_html
from .models import Reserva

#@admin.register(Reserva)
class ReservaAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'fecha', 'numero_personas', 'telefono', 'ver_detalles')
    search_fields = ('nombre', 'fecha')

    def has_change_permission(self, request, obj=None):
        # Si el usuario no ha solicitado editar, no permite cambios
        if obj is not None and not request.GET.get('edit', False):  # Aquí le agregamos la lógica de que no se pue
da editar
            return False
        return super().has_change_permission(request, obj)

    def ver_detalles(self, obj):
         return format_html(
            ''
            '✏ Editar',
            obj.id
        )
    ver_detalles.short_description = 'Detalles'


admin.site.register(Reserva, ReservaAdmin)
```

Estética con Jazzmin

Finalmente, como mencionaste, estás utilizando **Jazzmin** para mejorar la apariencia de la interfaz de administración. Jazzmin es un paquete de Django que proporciona una apariencia moderna y atractiva para el panel de administración, además de incluir diversas funcionalidades de personalización.

Jazzmin no necesita configuración adicional en admin.py, ya que simplemente debes instalarlo y configurarlo en tu archivo settings.py:

```
 pipenv install django-jazzmin
```

Y en settings.py, lo añades a INSTALLED_APPS:

```
 INSTALLED_APPS = [
    'jazzmin',  # Agregar jazzmin antes de 'django.contrib.admin'
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    ...
]
```

Y finalmene podremos ver nuestro admin, totalmente funcional y con una estética moderna

Resumen

- La estructura de los modelos conversacionales es esencial, centrándose en el modelo que se utilizará para procesar y generar respuestas en el chatbot.
- La integración del flujo conversacional en los modelos implica la conexión del diálogo del usuario con la lógica subyacente del chatbot, permitiendo una interacción más dinámica.
- Es fundamental entender los archivos involucrados, como **Chatbot_logic.py**, que contiene el código esencial para la operación del chatbot, y **nlp_utils.py**, que proporciona utilidades para el procesamiento del lenguaje natural.
- Se debe destacar cómo cada parte del código contribuye al funcionamiento general del chatbot, facilitando la interacción del usuario con el sistema.
- Otros archivos importantes están diseñados para complementar y soportar la lógica y funcionamiento del chatbot en un entorno Django.
- La administración y visualización de la reserva se gestiona a través de la creación de un superusuario, quien puede controlar y monitorizar las interacciones del chatbot.
- El archivo **admin.py** se encarga del manejo de la interfaz de administración, permitiendo la visualización y edición de las reservas.
- Se incorpora lógica de permisos de edición condicional para controlar quién puede modificar ciertos detalles en el sistema administrativo.
- Un botón de "Ver detalles" con enlace a la edición mejora la navegabilidad, proporcionando acceso rápido a la información detallada de las reservas.
- La estética se mejora usando **Jazzmin**, que ofrece una interfaz más amigable y visualmente atractiva para los administradores.

Actividades prácticas

Integración de modelos en el panel de administración de Django

En esta actividad práctica, aplicarás lo aprendido para integrar y gestionar modelos en el panel de administración de Django. Deberás configurar un modelo llamado 'Mensaje' para ser administrado en Django, añadiendo la lógica necesaria para crear un botón de 'Ver detalles' que permita editar elementos condicionados a un parámetro en la URL, similar al modelo de 'Reserva' que se aprendió en esta lección. Asegúrate de seguir los pasos proporcionados en la unidad y de integrar correctamente la apariencia moderna utilizando Jazzmin.

Pasos a seguir:

1. Crea un modelo llamado 'Mensaje' con campos básicos como 'contenido', 'fecha_envio' y 'usuario'.
2. Registra el modelo en el archivo 'admin.py', siguiendo la estructura aprendida en la unidad para la clase 'ReservaAdmin'.
3. Implementa la lógica del método 'has_change_permission' para controlar los permisos de edición basados en un parámetro de la URL.
4. Añade un método 'ver_detalles' que genere un enlace para editar el 'Mensaje', similar al de la unidad.
5. Asegúrate de incluir Jazzmin en el proyecto y verifica que tu panel de administración tenga la apariencia moderna esperada.
6. Realiza una prueba para asegurarte de que el enlace de edición únicamente funcione con el parámetro correcto.

1. Implementa la clase de administración 'MensajeAdmin' para el modelo 'Mensaje'. Asegúrate de que incluya los elementos de visualización de campos y búsqueda.

2. Permite que los administradores puedan editar mensajes solo si el parámetro 'edit=True' está presente en la URL.

3. Describe cómo integrar Jazzmin para mejorar la estética del panel de administración de Django. Incluye los pasos necesarios.

4. Explica cómo el método 'ver_detalles' proporciona acceso a las opciones de edición condicionalmente.

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

Recursos

Enlaces de Interés

Twilio
https://www.twilio.com/es-mx

La Plataforma de interacción con el cliente de Twilio combina potentes API de comunicaciones con IA y datos propios.

Cómo hacer registro
https://docs.python.org/es/3.13/howto/logging.html

Este apartado de la documentación oficial permite realizar un registro del programa ejecutandose