



PROYECTO SGE

CFGS Desarrollo de Aplicaciones Multiplataforma
Informática y Comunicaciones

APLICACIÓN CON FASTAPI

Año: 2025

Fecha de presentación: 30/05/2025

Nombre y Apellidos: Mario Gangoso Ferreras

Email: mario.ganfer@educa.jcyl.es

1. Investigación conceptos despliegue aplicaciones:

- Utilizando este enlace, indicar las definiciones que se piden a continuación:

https://raul-profesor.github.io/Despliegue/debian_teorias/

- VPS – Sus siglas significan Virtual Private Server y es un tipo de alojamiento web que proporciona una partición privada y recursos dedicados dentro de un servidor compartidos con otros usuarios.
- SSH – Es un protocolo de red que permite acceder y administrar sistemas de forma remota a través de una conexión cifrada.
- Servidor web – Sistema informático que aloja y sirve contenido web a través de internet.
- Protocolo HTTP, métodos - El protocolo de transferencia de hipertexto (HTTP, Hypertext Transfer Protocol) es el motor que da vida a Internet, ya que es la base para la web.

Metodos:

- **GET:** se utiliza para solicitar cualquier tipo de información o recurso al servidor. Cada vez que se pulsa sobre un enlace o se teclea directamente a una URL se usa este comando. Como resultado, el servidor HTTP enviará el recurso correspondiente.
- **HEAD:** se utiliza para solicitar información sobre el recurso: su tamaño, su tipo, su fecha de modificación... Es usado por los gestores de cachés de páginas o los servidores proxy, para conocer cuándo es necesario actualizar la copia que se mantiene del recurso. Con HEAD se podrá comprobar la última fecha de modificación de un recurso antes de traer una nueva copia del mismo.
- **POST:** sirve para enviar información al servidor, por ejemplo, los datos contenidos en un formulario. El servidor pasará esta información a un proceso encargado de su tratamiento.
- **OPTIONS:** Devuelve los métodos HTTP que el servidor soporta para una URL específica. Esto puede ser utilizado para comprobar la funcionalidad de un servidor web mediante petición en lugar de un recurso específico.
- **DELETE:** sirve para eliminar un recurso especificado en la URL, aunque pocas veces sera permitido por un servidor web.
- **TRACE:** comando que permite hacer un sondeo para saber todos los dispositivos de la red por los que pasa nuestra petición. Así podremos descubrir si la petición pasa a través dispositivos

intermedios o proxys antes de llegar al servidor Web.

- **PUT:** puede verse como el comando inverso a GET. Nos permite escribir datos en el servidor o, lo que es lo mismo, poner un recurso en la URL que se especifique. Si el recurso no existe lo crea sino lo reemplaza. La diferencia con POST puede ser algo confusa; mientras que POST está orientado a la creación de nuevos contenidos, PUT está más orientado a la actualización de los mismos (aunque también podría crearlos).

- Apache, nginx – Son dos servidores de código abierto que se utilizan para entregar contenido web a los usuarios a través de internet.

- Indicar 3 lenguajes de programación del lado del servidor
PHP, Python, Ruby.

- Utilizando este enlace, indicar las siguientes definiciones:

<https://openwebinars.net/blog/tensorflow-keras-fundamentos/>

- Tensorflow - Es una biblioteca open source desarrollada por Google brain para llevar a cabo tareas de machine learning y deep learning. Es una de las librerías de aprendizaje automático más utilizadas a nivel industrial debido a su flexibilidad y escalabilidad.

- Keras - Es una biblioteca de código abierto para el aprendizaje automático y el deep learning. Fue desarrollada por François Chollet y actualmente es mantenida por Google.

Se caracteriza por ser una biblioteca de alto nivel, fácil de usar y flexible, que permite a los desarrolladores crear y entrenar modelos de aprendizaje automático de manera rápida y eficiente.

- Buscar las siguientes definiciones en este enlace:

<https://la.mathworks.com/discovery/lstm.html>

- Modelo LSTM - Son redes neuronales recurrentes (RNN) especializadas diseñadas para procesar y recordar datos secuenciales, como series de tiempo o texto.

- Buscar las definiciones de:

- numpy - Es una biblioteca de Python de código abierto fundamental para la computación científica, especialmente en áreas como ciencia de datos, ingeniería y matemáticas. Permite trabajar

con objetos llamados arreglos multidimensionales, que son como matrices, y ofrece una gran cantidad de funciones matemáticas de alto nivel para operar con ellos de manera eficiente.

- matplotlib - Es una biblioteca de visualización de datos en Python que permite crear gráficos de diferentes tipos, como histogramas, diagramas de barras, gráficos de dispersión, entre otros, con unas pocas líneas de código.
- Google Colab - Es un servicio gratuito basado en la nube que permite escribir y ejecutar código Python, especialmente útil para tareas de aprendizaje automático y análisis de datos.

2. Creación de un modelo LSTM con TensorFlow:

- Utilizando este enlace, realizar lo que se pide a continuación:

<https://github.com/josejuansanchez/modelo-lstm-tensorflow>

- Leer el apartado: **modelo-lstm-tensorflow**
- Acceder al **notebook** del **contenido del repositorio** y comprender cómo funciona Google Colab
- En otra pestaña del navegador, abrir Google Colab: <https://colab.research.google.com/> y realizar lo siguiente:
 - Crear un cuaderno nuevo y cambiarle el nombre: `modeloLSTM_SGE2425.ipynb`
 - El siguiente enlace lleva a mi cuaderno.

<https://colab.research.google.com/drive/1kSQfcLPCpA-oB6knD8-Y8ursZFHzHJSi?usp=sharing>

Segunda parte del proyecto

1. Descripción general del proyecto

Desarrollar una API REST que permita clasificar preguntas automáticamente en distintas categorías predefinidas: OK, AYUDA y SERVICIO_TECNICO, utilizando un modelo de machine learning entrenado previamente con TensorFlow.

Además de la clasificación, se ha integrado un sistema de autenticación mediante tokens JWT, lo que permite proteger determinados endpoints y garantizar que solo usuarios autenticados puedan acceder a ciertas funcionalidades, como registrar nuevas predicciones en la base de datos.

El proyecto también ha incluido:

- Conexión e integración con una base de datos PostgreSQL.
- Persistencia de las predicciones realizadas.
- Implementación de pruebas unitarias para asegurar la funcionalidad básica del sistema.
- Uso de Swagger UI para la documentación y prueba interactiva de la API.

1.1. Entorno de trabajo (tecnologías de desarrollo y herramientas)

- Lenguaje de programación: Python
- Framework backend: FastAPI (para construcción de la API REST)
- Machine Learning: TensorFlow (modelo entrenado previamente y cargado en tiempo de ejecución)
- ORM y Base de datos: SQLAlchemy para la conexión con una base de datos PostgreSQL
- Autenticación: JWT (JSON Web Token), usando la librería python-jose
- Pruebas: Pytest
- Contenedores y despliegue: Docker y Docker Compose para contenerizar el proyecto y los servicios (como la base de datos)
- Editor de código: Visual Studio Code
- Documentación automática: Swagger UI, generado automáticamente por FastAPI
- Control de versiones: Git

2. Documentación técnica: análisis, diseño, implementación, pruebas y despliegue

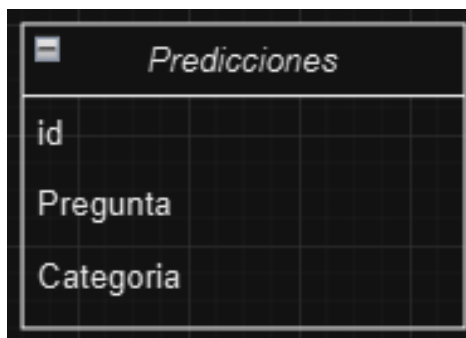
2.1. Análisis del sistema (funcionalidades básicas de la aplicación)

La API ofrece las siguientes funcionalidades principales:

- POST /predict: Clasifica una pregunta enviada en el cuerpo del request y la guarda en la base de datos. Esta ruta está protegida por autenticación JWT.
- GET /ping: Ruta pública para comprobar si la API está en funcionamiento.
- GET /prueba: Ruta de prueba sin lógica adicional, útil para testear conectividad.

- POST /login: Permite a un usuario autenticarse (con credenciales fijas o desde base de datos, según implementación) y obtener un token JWT para acceder a rutas protegidas.
- Las predicciones realizadas se almacenan con la pregunta original y la categoría asignada en la tabla predicciones de la base de datos PostgreSQL.

2.2. Diseño de la base de datos



Predicciones		
id		
Pregunta		
Categoría		

2.3. Implementación

Requirements

El proyecto ha sido desarrollado en Python utilizando el framework FastAPI para la creación de una API REST que permite clasificar preguntas mediante un modelo de machine learning. Se empleó TensorFlow para la carga y ejecución del modelo, y SQLAlchemy junto a PostgreSQL para la gestión de la base de datos. La validación de datos se realizó con Pydantic, y la autenticación de usuarios mediante tokens JWT con la librería python-jose. Para las pruebas unitarias se usaron pytest y httpx. El servidor Uvicorn permite ejecutar la API localmente, y todo el entorno de trabajo ha sido gestionado en Visual Studio Code sobre un entorno virtual.

```
requirements.txt
1  fastapi
2  uvicorn
3  sqlalchemy
4  pydantic
5  tensorflow
6
7  # Requisitos para testing
8  pytest
9  httpx
10
11 #Requisitos para JWT
12 python-jose[cryptography]
```

docker-compose.yml

Para facilitar la gestión del entorno de desarrollo y despliegue, se utilizó Docker mediante un archivo `docker-compose.yml`. Este define dos servicios: una base de datos PostgreSQL que actúa como backend de almacenamiento, y PgAdmin como herramienta de administración visual para PostgreSQL. Ambos servicios están conectados a través de una red personalizada y configurados con variables de entorno para facilitar el acceso y la persistencia de datos mediante volúmenes.

```
version: "3.8"

services:
  db:
    image: postgres:15
    container_name: fastapi_postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: fastapi_user
      POSTGRES_PASSWORD: fastapi_pass
      POSTGRES_DB: fastapi_db
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - fastapi_net

  pgadmin:
    image: dpage/pgadmin4
    container_name: fastapi_pgadmin
    restart: unless-stopped
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "5050:80"
    depends_on:
      - db
    networks:
      - fastapi_net

volumes:
  postgres_data:

networks:
  fastapi_net:
```

Main

El archivo main.py actúa como punto de entrada principal de la API. Se encarga de configurar y lanzar la aplicación FastAPI, estableciendo metadatos como el título, la descripción y la versión del servicio. También crea las tablas necesarias en la base de datos a partir de los modelos definidos con SQLAlchemy, e incluye los routers de las rutas principales: predict para realizar predicciones de clasificación de preguntas, y auth para el sistema de autenticación mediante JWT. Además, define una ruta raíz (/) que devuelve un mensaje simple, y configura el servidor de desarrollo con Uvicorn para ejecutar la API en el puerto 8000.

```
from fastapi import FastAPI
from app.routers import predict
from app.db.database import Base, engine
from app.db import models
import uvicorn
from app.routers import predict, auth

app = FastAPI(
    title="API de Clasificación de Preguntas",
    description="Clasifica preguntas en categorías: OK, AYUDA, SERVICIO_TECNICO",
    version="1.0"
)

Base.metadata.create_all(bind=engine)

app.include_router(predict.router)

# Ruta raíz para evitar 404 en /
@app.get("/")
async def root():
    return {"message": "API de Clasificación de Preguntas está funcionando"}

app.include_router(auth.router)
app.include_router(predict.router)

if __name__ == "__main__":
    uvicorn.run("main:app", port=8000, reload=True)
```


App/db/Database

El archivo database.py configura la conexión con la base de datos PostgreSQL usando SQLAlchemy. Define la URL de conexión con las credenciales y parámetros necesarios para acceder a la base de datos, y crea un motor (engine) que gestiona la conexión con el servidor. Además, se define SessionLocal, que permite crear sesiones para interactuar con la base de datos dentro de las rutas de la API. Finalmente, se declara Base, la clase base de la que heredarán todos los modelos de datos del proyecto para que puedan ser registrados correctamente en la base de datos.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

SQLALCHEMY_DATABASE_URL = "postgresql://fastapi_user:fastapi_pass@localhost:5432/fastapi_db"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

App/db/Models

El archivo models.py define el modelo de datos Prediccion, que representa la tabla predicciones en la base de datos PostgreSQL. Este modelo hereda de Base, que está configurada en database.py. La tabla contiene tres columnas: id (clave primaria y autoincremental), pregunta (texto de la consulta enviada por el usuario) y categoria (la clase o tipo al que pertenece la pregunta, como OK, AYUDA o SERVICIO_TECNICO). Este modelo es fundamental para almacenar las predicciones generadas por el sistema de clasificación.

```
from sqlalchemy import Column, Integer, String
from app.db.database import Base

class Prediccion(Base):
    __tablename__ = "predicciones"

    id = Column(Integer, primary_key=True, index=True)
    pregunta = Column(String, nullable=False)
    categoria = Column(String, nullable=False)
```

App/auth/ jwt_handler

El archivo `jwt_handler.py` implementa la lógica para la creación y verificación de tokens JWT (JSON Web Tokens), fundamentales para la autenticación de usuarios en la API. Se utiliza la librería `python-jose` con el algoritmo `HS256` y una clave secreta (que debería almacenarse de forma segura en producción). La función `create_access_token` genera un token con datos del usuario y una expiración de 30 minutos, mientras que `verify_token` se encarga de decodificar y validar el token recibido, asegurando que sea auténtico y no haya expirado.

```
from jose import jwt, JWTError
from datetime import datetime, timedelta

# Clave secreta (no debe estar hardcodeada en producción)
SECRET_KEY = "supersecreta123"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    token = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return token

def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        return None
```

App/routers/auth

El archivo auth.py define un endpoint /login para la autenticación de usuarios utilizando FastAPI y Pydantic. La autenticación se basa en un diccionario simulado de usuarios con sus contraseñas en memoria (fake_users_db). Cuando un usuario envía sus credenciales, se verifica la correspondencia y, si son válidas, se genera un token JWT mediante la función create_access_token del módulo jwt_handler. Este token es devuelto como respuesta para que el cliente lo use en peticiones autenticadas. En caso de credenciales incorrectas, se devuelve un error 401 Unauthorized

```
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from app.auth.jwt_handler import create_access_token

router = APIRouter()

# Simulación de usuarios (en memoria)
fake_users_db = {
    "admin": "admin123"
}

class UserLogin(BaseModel):
    username: str
    password: str

@router.post("/login")
def login(user: UserLogin):
    if user.username in fake_users_db and fake_users_db[user.username] == user.password:
        token = create_access_token({"sub": user.username})
        return {"access_token": token, "token_type": "bearer"}
    raise HTTPException(status_code=401, detail="Credenciales inválidas")
```

App/router/predict

Este módulo define el router /predict usando FastAPI. Aquí se implementa la lógica de autenticación con JWT mediante el esquema HTTPBearer para proteger la ruta de predicción.

- Autenticación:

La función get_current_user extrae y verifica el token JWT recibido en la cabecera Authorization. Si el token es inválido o ha expirado, lanza un error 401. Esto asegura que solo usuarios autenticados puedan usar la API.

- Dependencias:

Se usa Depends para inyectar la sesión de base de datos y la validación del usuario en el

endpoint.

- Endpoint /predict:
Recibe un JSON con la pregunta, valida el token, y llama a la función `predecir_categoria` para obtener la categoría usando un modelo TensorFlow previamente entrenado. Después, guarda la predicción en la base de datos. En caso de error en la predicción o falta de clasificación, responde con errores HTTP apropiados.
- Rutas adicionales:
`/ping` para verificar que la API está activa, y `/prueba` como ruta auxiliar de prueba.

Este diseño modular con dependencias y manejo de errores facilita la extensión y mantenimiento de la API, mientras protege las operaciones sensibles con autenticación.

```
router = APIRouter()

security = HTTPBearer()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

def get_current_user(token: HTTPAuthorizationCredentials = Depends(security)):
    user = verify_token(token.credentials)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Token inválido o expirado",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return user

@router.get("/ping")
def ping():
    return {"message": "API funcionando correctamente"}

@router.get("/prueba")
def prueba():
    return {"message": "Esta es una ruta de prueba"}

@router.post("/predict")
def predict(pregunta: PreguntaRequest, db: Session = Depends(get_db), user: dict = Depends(get_current_user)):
    try:
        categoria = predecir_categoria(pregunta.pregunta)
    except Exception:
        raise HTTPException(status_code=500, detail="Error interno al predecir la categoría")

    if not categoria:
        raise HTTPException(status_code=400, detail="No se pudo clasificar la pregunta")

    nueva_prediccion = models.Prediccion(pregunta=pregunta.pregunta, categoria=categoria)
    db.add(nueva_prediccion)
    db.commit()
    db.refresh(nueva_prediccion)

    return {
        "pregunta": pregunta.pregunta,
        "categoria": categoria
    }
```

App/schemas/predict

El esquema PreguntaRequest está definido con Pydantic y representa la estructura de los datos que la API espera recibir para la clasificación de preguntas. Contiene un único campo pregunta de tipo cadena (str), con validación incorporada que exige que la pregunta tenga entre 5 y 200 caracteres. Esta validación ayuda a asegurar que las entradas sean apropiadas para el modelo de clasificación.

```
from pydantic import BaseModel, Field

class PreguntaRequest(BaseModel):
    #Mínimo 5 caracteres y máximo 200
    pregunta: str = Field(..., min_length=5, max_length=200, description="Pregunta a clasificar")
```

App/model

Esta sección del código se encarga de la carga y uso del modelo de clasificación basado en TensorFlow. Primero, se carga un modelo previamente entrenado desde el directorio saved_model/modelo.keras. Luego, se prepara un tokenizer con la clase TextVectorization de TensorFlow, que convierte las preguntas en secuencias numéricas basadas en un vocabulario cargado desde vocabulary.txt. Además, se leen las etiquetas de clasificación desde un archivo JSON (etiquetas.json) y se crea un diccionario inverso para mapear índices a etiquetas.

La función predecir_categoria recibe una pregunta en formato texto, la preprocesa usando el tokenizer, realiza una predicción con el modelo y devuelve la etiqueta correspondiente a la categoría más probable. Si no encuentra una etiqueta válida, devuelve "NO_ENTIENDO".

```
import tensorflow as tf
import numpy as np
import json

# Cargar modelo
model = tf.keras.models.load_model("saved_model/modelo.keras")

# Tokenizer
with open("saved_model/vocabulary.txt", "r") as f:
    vocabulary = [line.strip() for line in f]

tokenizer = tf.keras.layers.TextVectorization(output_mode="int", output_sequence_length=20)
tokenizer.set_vocabulary(vocabulary)

# Etiquetas
with open("saved_model/etiquetas.json", "r") as f:
    etiquetas = json.load(f)

index_to_label = {v: k for k, v in etiquetas.items()}

def predecir_categoria(pregunta: str) -> str:
    texto_preprocesado = tokenizer([pregunta])
    prediccion = model.predict(texto_preprocesado)
    indice = int(np.argmax(prediccion[0]))
    return index_to_label.get(indice, "NO_ENTIENDO")
```

Test/test_predict

Esta sección contiene pruebas unitarias para la API utilizando pytest y TestClient de FastAPI. Primero, se ajusta el sys.path para importar el módulo principal correctamente. Luego, se crea un cliente de pruebas para la aplicación FastAPI importada desde main.py.

Se definen tres tests para el endpoint /predict:

- test_predict_success: Envía una pregunta válida y verifica que la respuesta tenga código 200, contenga las claves categoria y pregunta, y que el texto de la pregunta en la respuesta coincida con el enviado.
- test_predict_empty: Prueba una petición con el campo pregunta vacío y espera un error de validación 422 generado por Pydantic.
- test_predict_missing_field: Envía un payload sin el campo pregunta y también espera un error 422 por falta de datos obligatorios.

Estas pruebas garantizan que el endpoint maneje correctamente solicitudes válidas y errores básicos de validación.

```
import sys
import os

# Añadir el directorio raíz del proyecto al sys.path
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))

from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_predict_success():
    payload = {
        "pregunta": "No puedo conectarme a internet"
    }
    response = client.post("/predict", json=payload)

    assert response.status_code == 200
    data = response.json()
    assert "categoria" in data
    assert "pregunta" in data
    assert data["pregunta"] == payload["pregunta"]

def test_predict_empty():
    payload = {"pregunta": ""}
    response = client.post("/predict", json=payload)
    assert response.status_code == 422 # Validación de Pydantic

def test_predict_missing_field():
    payload = {} # No se envía 'pregunta'
    response = client.post("/predict", json=payload)
    assert response.status_code == 422
```

2.4. Pruebas

Realizar y documentar un mínimo de 2 pruebas: una con swagger y otra con pgadmin para ver el contenido de la tabla de la base de datos

Prueba de login

<http://127.0.0.1:8000/docs>

Busca la ruta POST /login.

Pulsa sobre ella y haz clic en Try it out.

Rellena el formulario con las credenciales correctas: Admin y admin123

Haz clic en Execute.

POST /login Login

Parameters Cancel Reset

No parameters

Request body required application/json

```
{  "username": "admin",  "password": "admin123"}
```

Execute

En la respuesta, copiar el campo "access_token" que te devuelve el servidor.

200

Response body

```
{  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbSI6ImV4cCI6MTc0ODU0MzI5Mn0.Xkg3_A9vym6uGh9VfagNSj42Ro4ABdXQwfk4VrjVDQ",  "token_type": "bearer"}
```

Download

Hacer clic en el botón Authorize

API de Clasificación de Preguntas 1.0 OAS 3.1

/openapi.json

Clasifica preguntas en categorías: OK, AYUDA, SERVICIO_TECNICO

Authorize



default



GET /ping Ping



GET /prueba Prueba



POST /predict Predict



GET / Root



POST /login Login



En el campo que aparece, escribe el token con el formato: Bearer y el token copiado anteriormente.

Pulsa Authorize y luego Close.

Available authorizations



HTTPBearer (http, Bearer)

Value:

Bearer eyJhbGciOiJIUzI1NiIs

Authorize

Close

Aquí podemos ver que ha ido todo bien. Ahora podemos hacer cualquier pregunta al predict.

Available authorizations



HTTPBearer (http, Bearer)

Authorized

Value: *****

Logout

Close

Abrir en el navegador <http://localhost:5050>

Iniciar sesión con los datos que definiste en el docker-compose.yml

The image shows the pgAdmin login interface. It has a dark blue header with the 'pgAdmin' logo. Below the logo is the word 'Login'. There are two input fields: the first contains 'admin@example.com' and the second contains five dots representing a password. To the right of the password field is a link that says 'Forgotten your password?'. Below these fields is a dropdown menu currently showing 'Spanish'. At the bottom is a large blue button labeled 'Login'.

Agregar un nuevo servidor

- Haz clic derecho sobre "Servidores" (barra izquierda).
- Selecciona "Crear" → "Servidor...".
- Nombre: fastapi_postgres
- Pestaña Conexión
 - Rellena con los datos de tu base de datos definidos en docker-compose.yml:
 - .1.1. **Host / nombre del servidor:** db
 - .1.2. **Puerto:** 5432
 - .1.3. **Mantenimiento / Base de datos:** fastapi_db
 - .1.4. **Nombre de usuario:** fastapi_user
 - .1.5. **Contraseña:** fastapi_pass
 - .1.6. Marca la opción "**Guardar contraseña**"

Register - Servidor

General

Conexión

Parámetros

Túnel SSH

Avanzado

Post Connection SQL

Tags

Nombre/Dirección de servidor

db

Puerto

5432

Base de datos de mantenimiento

fastapi_db

Nombre de usuario

fastapi_user

Kerberos authentication?

☐

Contraseña

.....

¿Salvar contraseña?

☐

Rol

Servicio

i

?

X Cerrar

↺ Restaurar

💾 Salvar

Ya conectado:

Desplegar Servidores → fastapi_postgres → Bases de datos → fastapi_db → Esquemas → public → Tablas.

Aquí está la tabla predicciones y sus columnas.



2.5. Despliegue de la aplicación

Este proyecto se ejecuta en un servidor local, utilizando contenedores Docker para levantar tanto la base de datos PostgreSQL como la herramienta de administración pgAdmin. La API desarrollada con FastAPI se ejecuta localmente a través de uvicorn, permitiendo pruebas y desarrollo en un entorno controlado. Todo el sistema está preparado para ser desplegado fácilmente en un servidor remoto si se desea escalar o publicar la solución en producción.

Manuales

2.6. Manual de usuario:

Breve explicación de cómo se usa la API con capturas de pantalla

Aquí están las rutas principales que aparecen en Swagger.

API de Clasificación de Preguntas 1.0 OAS 3.1

/openapi.json

Clasifica preguntas en categorías: OK, AYUDA, SERVICIO_TECNICO

default

**GET****/ping** Ping**GET****/prueba** Prueba**POST****/predict** Predict**GET****/** Root

Endpoints de prueba abiertos para verificar que la API funciona tanto /ping como /prueba.

The screenshot shows a REST client interface for a GET request to `/ping`. The interface includes a 'Parameters' section with a 'Cancel' button, an 'Execute' button, and a 'Clear' button. Below the 'Responses' section, the 'Curl' command is displayed: `curl -X 'GET' \ 'http://127.0.0.1:8000/ping' \ -H 'accept: application/json'`. The 'Request URL' is `http://127.0.0.1:8000/ping`. The 'Server response' section shows a 'Code' of 200 and a 'Response body' of `{ "message": "API funcionando correctamente" }`. The 'Response body' is highlighted with a red box.

The screenshot shows a REST client interface for a GET request to `/prueba`. The interface includes a 'Parameters' section with a 'Cancel' button, an 'Execute' button, and a 'Clear' button. Below the 'Responses' section, the 'Curl' command is displayed: `curl -X 'GET' \ 'http://127.0.0.1:8000/prueba' \ -H 'accept: application/json'`. The 'Request URL' is `http://127.0.0.1:8000/prueba`. The 'Server response' section shows a 'Code' of 200 and a 'Response body' of `{ "message": "Esta es una ruta de prueba" }`. The 'Response body' is highlighted with a red box.

El /login permite simular un inicio de sesión con un usuario y contraseña que hemos insertado en el código y devuelve un Access_token.

POST

/login Login

^

Parameters

Cancel

Reset

No parameters

Request body required

application/json

▼

```
{  
  "username": "admin",  
  "password": "admin123"  
}
```

Execute

Clear

Aquí vemos el access_token

The screenshot displays a REST client interface with the following sections:

- Curl**: A dark box containing the curl command:

```
curl -X 'POST' \  
  'http://127.0.0.1:8000/login' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "username": "admin",  
    "password": "admin123"  
  }'
```
- Request URL**: A dark box containing the URL `http://127.0.0.1:8000/login`.
- Server response**: A section with two tabs: **Code** and **Details**. The **Code** tab is active, showing a status code of **200** and a **Response body**. The response body is highlighted with a red border and contains the JSON object:

```
{  
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZGIpbWV4ImV4cCI6MTc0ODUzNTIzM30.-9MCHmBZ_6bgYt6SBx-PcqtswnwTnZ-Amzu2qGEiAw",  
  "token_type": "bearer"  
}
```

To the right of the JSON are icons for copying and a **Download** button.
- Response headers**: A dark box at the bottom listing the headers:

```
content-length: 165  
content-type: application/json  
date: Thu, 29 May 2025 15:44:32 GMT  
server: uvicorn
```

Ahora debemos copiar el Access_token anterior e ir a la parte superior derecha que pone Authorize.

[Authorize](#)

default



GET

[/ping](#) Ping

GET

[/prueba](#) Prueba

POST

[/predict](#) Predict

GET

[/](#) Root

POST

[/login](#) Login

En esta pantalla vamos a poner Bearer y el Access token que tenemos copiado, damos a authorize y vemos que si funciona.

Available authorizations



HTTPBearer (http, Bearer)

Value:

Bearer eyJhbGciOiJIUzI1NiIs

[Authorize](#)[Close](#)

Ahora nos deja hacer preguntas y nos devuelve la categoría en /predict cuando antes no nos dejaba porque no teníamos acceso autorizado.

The screenshot displays a REST client interface with the following sections:



- POST /predict Predict**: The endpoint and method.
- Parameters**: A section with "No parameters" and buttons for "Cancel" and "Reset".
- Request body required**: A dropdown menu set to "application/json".
- Request body**: A text area containing the JSON payload:

```
{  "pregunta": "Necesito cambiar una rueda del coche"}
```
- Execute**: A blue button to send the request.
- Clear**: A button to clear the request body.
- Curl**: A section showing the equivalent curl command:

```
curl -X 'POST' \  'http://127.0.0.1:8000/predict' \  -H 'accept: application/json' \  -H 'Content-Type: application/json' \  -d '{  "pregunta": "Necesito cambiar una rueda del coche"  }'
```
- Request URL**: A text field containing `http://127.0.0.1:8000/predict`.
- Server response**: A table showing the response details.

Code	Details
200	<div>Response body<pre>{ "pregunta": "Necesito cambiar una rueda del coche", "categoria": "SERVICIO_TECNICO"}</pre></div> <div> Download</div>

Nos mostraría este mensaje si no está autorizada la sesión

Code	Details
403 <i>Undocumented</i>	Error: Forbidden Response body <pre>{ "detail": "Not authenticated" }</pre>  

2.7. Manual de instalación:

Crear y activar el entorno virtual

```
Python -m venv venv
```

```
Venv\Scripts\activate
```

Instalar dependencias

```
Pip install -r requirements.txt y después de requirements me pide pip install psycpg2-binary
```

Levantar contenedores de docker

```
docker-compose up -d
```

Ejecutar la API

```
python main.py
```

3. Conclusiones

- Durante el desarrollo de esta aplicación se presentaron algunas dificultades, especialmente en la integración del sistema de autenticación con JWT y la protección de rutas privadas, como la del endpoint /predict. También resultó algo complejo configurar correctamente la base de datos PostgreSQL con Docker y lograr que FastAPI pudiera comunicarse con ella sin problemas.
- Otra dificultad fue la gestión de permisos y cabeceras a la hora de realizar pruebas en Swagger, ya que requería entender bien cómo se pasaban los tokens Bearer para que la API autorizara las peticiones correctamente.
- A pesar de estos retos, el grado de satisfacción con el trabajo realizado es alto, ya que se logró implementar una API funcional y segura que permite clasificar preguntas usando un modelo de machine learning entrenado previamente, además de dejar la aplicación lista para futuras mejoras o despliegues.
- Durante este proyecto se ha afianzado el uso de FastAPI, SQLAlchemy, JWT y la integración de modelos de TensorFlow en APIs REST. Además, se ha aprendido a trabajar con entornos Dockerizados y a realizar pruebas automatizadas con pytest.

Ampliación

Una posible ampliación del proyecto sería permitir el registro de nuevos usuarios, añadiendo persistencia en base de datos y roles de usuario.

4. GitHub

<https://github.com/MarioGangosoRibera/ApiChatbotPython.git>

5. Bibliografía

- **Documentación oficial de FastAPI**
<https://fastapi.tiangolo.com/>
- **Documentación de SQLAlchemy**
<https://docs.sqlalchemy.org/>
- **Documentación de Pydantic**
<https://docs.pydantic.dev/>
- **Documentación de TensorFlow**
https://www.tensorflow.org/api_docs
- **Documentación de PostgreSQL**
<https://www.postgresql.org/docs/>
- **JWT (JSON Web Tokens) - jose package**
<https://python-jose.readthedocs.io/en/latest/>
- **Chatgpt**