Detailed Explanation of the Sentiment Analysis API Project

This guide will help you **understand every aspect** of the project in detail.

---

1. Setup We Made to Start the Project

To start the project, we followed these steps:

🔷 Project Folder Structure

```
SentimentAnalysisAPI/
|── main.py          # FastAPI application
|── test_performance.py  # Performance testing script
|── requirements.txt    # Dependencies
|── metrics.csv        # Stores API request logs
|── reviews.txt        # Input test cases
|── venv/             # Virtual environment
```

2. What Libraries or Packages We Have Used

Package	**Purpose**
FastAPI	API framework for creating REST endpoints
Uvicorn	ASGI server for running FastAPI
SQLAlchemy	ORM for database integration
TextBlob	Sentiment analysis tool
PyJWT	Handles JWT authentication
Pydantic	Data validation for API requests
Httpx	Sends async HTTP requests for testing
Pytest	Runs unit tests
pytest-asyncio	Allows async testing
Passlib & Bcrypt	Used for password hashing (future security implementation)
CSV	Handles logging request data in metrics.csv
Asyncio	Enables handling concurrent API calls

---

3. What is venv and How It Works

🔷 What is venv?

venv (Virtual Environment) is used to create **an isolated Python environment** where project dependencies are installed. This prevents conflicts with globally installed Python packages.

🔷 How venv Works

1. **Creates a new folder venv/**, which stores a separate Python interpreter and dependencies.
2. **When activated**, all Python commands use this virtual environment instead of the system-wide Python.
3. **Deactivating venv** restores the system Python environment: **deactivate**

Detailed Explanation of main.py in the Sentiment Analysis API

---

1. Setup & Initial Configuration

🔷 Importing Required Modules

```
import logging

import time

import csv

import uuid

from fastapi import FastAPI, HTTPException, Depends, Request

from pydantic import BaseModel

from sqlalchemy import create_engine, Column, Integer, String

from sqlalchemy.orm import sessionmaker, declarative_base

from textblob import TextBlob

import jwt

import datetime

from jwt.exceptions import ExpiredSignatureError, InvalidTokenError

from passlib.context import CryptContext
```

🔷 **Purpose of Each Import:**

- **logging** → For structured logs to monitor API activity.
- **time** → Used to calculate API execution time.
- **csv** → Used to log metrics in metrics.csv.
- **uuid** → Generates unique request IDs.
- **FastAPI components** → Handles API routes, authentication, and request processing.
- **Pydantic** → Validates incoming JSON request bodies.
- **SQLAlchemy** → Handles database operations.
- **TextBlob** → Performs sentiment analysis on review text.
- **JWT (PyJWT)** → Manages authentication tokens.
- **datetime** → Used for timestamping logs and managing token expiration.
- **passlib** → Secure password hashing (not yet used but useful for future security improvements).

---

2. Setting Up Logging

```
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

logger = logging.getLogger(__name__)
```

- Configures structured logging for monitoring requests and responses.
- **Example Log Output:**
- 2025-01-31 20:00:11,311 - INFO - Request ID: 1234abcd | Sentiment Analysis: "Great product" -> Positive (90%) | Execution Time: 2.1s

---

3. Initializing FastAPI and Database

🔷 **FastAPI App Instance**

```
app = FastAPI()
```

- Creates the FastAPI app instance to define routes.

🔷 **Database Configuration (SQLite with SQLAlchemy)**

```
DATABASE_URL = "sqlite:///./database.db"

engine = create_engine(DATABASE_URL, echo=True, connect_args={"check_same_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

```
Base = declarative_base()
```

- **Uses SQLite as the database.**
- **Creates an ORM session (SessionLocal) to interact with the database.**
- **echo=True** → Logs SQL queries for debugging.

🔷 **Defining Database Model (Feedback Table)**

```
class Feedback(Base):
    __tablename__ = "feedback"
    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer)
    review_text = Column(String)
    sentiment = Column(String)
    confidence = Column(Integer)
```

- **Stores sentiment analysis results in SQLite.**
- **Columns:**
  - id → Auto-increment primary key.
  - user_id → Stores the user ID of the reviewer.
  - review_text → Stores the review text.
  - sentiment → Stores the sentiment result (positive, negative, neutral).
  - confidence → Stores confidence percentage based on sentiment polarity.

```
Base.metadata.create_all(bind=engine)
logger.info("Database setup complete")
```

- Creates the feedback table in SQLite.

---

4. Defining API Models with Pydantic

```
class Review(BaseModel):
    user_id: int
    review_text: str
class UserLogin(BaseModel):
    username: str
    password: str
```

- **Review Model:** Defines request structure for sentiment analysis API.
- **UserLogin Model:** Defines request structure for login authentication.

---

5. JWT Token Authentication

🔷 JWT Secret Key & Algorithm

```
SECRET_KEY = "your_secret_key_here"

ALGORITHM = "HS256"
```

- Used for **signing JWT tokens** to prevent tampering.

🔷 Function to Create JWT Token

```
def create_jwt_token(data: dict, expires_delta: int = 60):

    expire = datetime.datetime.utcnow() + datetime.timedelta(minutes=expires_delta)

    to_encode = data.copy()

    to_encode.update({"exp": expire})

    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

- **Generates JWT token** that expires after 60 minutes.
- **Includes exp (expiration time) to enhance security**.

🔷 Function to Verify JWT Token

```
def verify_jwt_token(token: str):

    try:

        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])

        return payload

    except ExpiredSignatureError:

        raise HTTPException(status_code=401, detail="Token has expired")

    except InvalidTokenError:

        raise HTTPException(status_code=401, detail="Invalid token")
```

- **Decodes JWT and verifies validity**.
- **Raises error if token is expired or invalid**.

---

6. Login Route to Generate JWT Token

```
@app.post("/login")
def login(user: UserLogin):
    if user.username == "admin" and user.password == "password":
        token = create_jwt_token({"sub": user.username})
        return {"access_token": token, "token_type": "bearer"}
    raise HTTPException(status_code=401, detail="Invalid username or password")
```

- **Hardcoded credentials** (admin/password) for testing.
- **Returns JWT token** to be used in API requests.

---

7. Logging API Metrics in metrics.csv

```
def log_metrics(request_id, user_id, review_text, sentiment, confidence, execution_time):
    with open("metrics.csv", mode="a", newline="") as file:
        writer = csv.writer(file)
        writer.writerow([
            datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            request_id,
            user_id,
            review_text,
            sentiment,
            confidence,
            execution_time
        ])
```

- **Appends request logs to metrics.csv** for performance tracking.

---

8. Sentiment Analysis Endpoint (/analyze/)

```
@app.post("/analyze/")
def analyze_sentiment(review: Review, request: Request):
    start_time = time.time()
    request_id = str(uuid.uuid4())

    authorization: str = request.headers.get("Authorization")
    if not authorization:
        raise HTTPException(status_code=401, detail="Authorization token is missing")

    token = authorization.split(" ")[1]
    user_payload = verify_jwt_token(token)

    analysis = TextBlob(review.review_text)
    sentiment = "positive" if analysis.sentiment.polarity > 0 else "negative" if analysis.sentiment.polarity < 0 else "neutral"
    confidence = abs(analysis.sentiment.polarity) * 100

    db = SessionLocal()
    feedback = Feedback(user_id=review.user_id, review_text=review.review_text, sentiment=sentiment, confidence=confidence)
    db.add(feedback)
    db.commit()
    db.refresh(feedback)
    db.close()

    execution_time = round(time.time() - start_time, 4)
    log_metrics(request_id, review.user_id, review.review_text, sentiment, confidence, execution_time)
    logger.info(f"Request ID: {request_id} | Sentiment Analysis: {review.review_text} -> {sentiment} ({confidence}%) | Execution Time: {execution_time}s")


    return {"request_id": request_id, "user_id": review.user_id, "sentiment": sentiment, "confidence": confidence, "execution_time": execution_time}
```

- **Validates JWT token** before processing request.
- **Uses TextBlob to determine sentiment polarity**.
- **Saves results to SQLite** and logs execution time.
- **Returns analysis result to user**.

---

9. Root Endpoint

```
@app.get("/")
def read_root():
    return {"message": "Welcome to Sentiment Analysis API"}
```

- **Simple endpoint to check API status**.

---

5. Explanation of test_performance.py Line by Line

```
import asyncio
import httpx
import time
import csv
import pytest
```

- **Imports modules for async requests & testing**.

```
@pytest.mark.asyncio
async def test_performance():
    async with httpx.AsyncClient() as client:
        response = await client.post("http://127.0.0.1:8000/analyze/", json={"user_id": 1, "review_text": "Great product!"})
        assert response.status_code == 200
```

- **Sends multiple API requests asynchronously**.
- **Asserts that all responses return 200 OK**.

6. How API Creation Works

1. **FastAPI initializes** the web service.
2. **Routes (@app.post) define API endpoints**.
3. **Dependency injection (Depends) handles authentication**.
4. **The API processes user input, applies sentiment analysis, and returns results**.

7. How JWT Tokens Were Implemented

1. **User logs in via /login**.
2. **A JWT token is created** using jwt.encode().
3. **This token is sent with every API request**.
4. **The API verifies the token before processing requests**.

8. How We Calculated Polarity and Determined Sentiment

- **Used TextBlob**:
- analysis = TextBlob(review.review_text).sentiment.polarity
- **Polarity Scale**:
  - **> 0 → Positive sentiment**
  - **< 0 → Negative sentiment**
  - **= 0 → Neutral sentiment**

9. How We Save Results in metrics.csv

- **Each API request logs its execution time, sentiment, and confidence score**.
- **Data is stored in metrics.csv** to track system performance.
- **Example CSV structure**:
- timestamp, request_id, user_id, review_text, sentiment, confidence_score, execution_time
- 2025-01-31 20:00:11, 92cde392-d5b5-4663, 1, "I love this!", positive, 92%, 3.5s

TextBlob :

- TextBlob is a Python library for processing textual data.
- It provides a consistent API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, and more.

Sentiment Analysis using Textblob

- Sentiment Analysis can assist us in determining the mood and feelings of the general public as well as obtaining useful information about the setting.
- Sentiment Analysis is the process of assessing data and categorizing it according to the needs.
- The polarity and subjectivity of a statement are returned by TextBlob.
- The range of polarity is [-1,1], with -1 indicating a negative sentiment and 1 indicating a positive sentiment.
- Negative words are used to change the polarity of a sentence. Semantic labels in TextBlob aid in fine-grained analysis.
- Emoticons, exclamation marks, and emojis, for example. subjectivity falls under the numeric range of [0,1].
- The degree of personal opinion and factual information in a text is measured by subjectivity.
- Because of the text's heightened subjectivity, it contains personal opinion rather than factual information.
- There's one more setting in TextBlob: intensity. The 'intensity' is used by TextBlob to calculate subjectivity. The intensity of a word influences whether it modifies the next word. Adverbs are used as modifiers in English.
- By providing an input sentence, the TextBlob's sentiment property returns a named tuple with polarity and subjectivity scores.
- The polarity score ranges from -1.0 to 1.0 and the subjectivity ranges from 0.0 to 1.0 where 0.0 is an objective statement and 1 is a subjective statement.
- Example : Input : **my_sentence.sentiment**
  Output : **Sentiment(polarity=0.75, subjectivity=0.95)**

FastAPI

- FastAPI is a modern web framework that is relatively fast and used for building APIs with Python 3.7+ based on standard Python-type hints.
- FastAPI also assists us in automatically producing documentation for our web service so that other developers can quickly understand how to use it.
- This documentation simplifies testing web service to understand what data it requires and what it offers.
- FastAPI has many features like it offers significant speed for development and also reduces human errors in the code.
- It is easy to learn and is completely production-ready. FastAPI is fully compatible with well-known standards of APIs.

Features of FastAPI

- **Automatic Documentation:** FastAPI generates interactive API documentation automatically using the OpenAPI standard. You can access this documentation by visiting a specific endpoint in your application, which makes it incredibly easy to understand and test your API without having to write extensive documentation manually.
- **Python Type Hints**: One of FastAPI's standout features is its use of Python-type hints. By annotating function parameters and return types with type hints, you not only improve code readability but also enable FastAPI to automatically validate incoming data and generate accurate API documentation. This feature makes your code less error-prone and more self-documenting.
- **Data Validation:** FastAPI uses Pydantic models for data validation. You can define your data models using Pydantic's schema and validation capabilities. This ensures incoming data is automatically validated, serialized, and deserialized, reducing the risk of handling invalid data in your application.
- **Asynchronous Support:** With the rise of asynchronous programming in Python, FastAPI fully embraces asynchronous operations. You can use Python's async and await keywords to write asynchronous endpoints, making it well-suited for handling I/O-bound tasks and improving the overall responsiveness of your application.
- **Dependency Injection:** FastAPI supports dependency injection, allowing you to declare dependencies for your endpoints. This helps in keeping your code modular, testable, and maintainable. You can seamlessly inject dependencies like database connections, authentication, and more into your routes.
- **Security Features:** FastAPI includes various security features out of the box, such as support for OAuth2, JWT (JSON Web Tokens), and automatic validation of request data to prevent common security vulnerabilities like SQL injection and cross-site scripting (XSS) attacks.

Role of Pydantic in FastAPI

- Pydantic is a Python library that is commonly used with FastAPI. It plays a crucial role in FastAPI applications by providing data validation, parsing, and serialization capabilities.
- Specifically, Pydantic is used in FastAPI. Pydantic is a Python library that shines when it comes to data validation and parsing.
- In FastAPI, Pydantic plays a crucial role in several key areas:

Validation Data with Pydantic

- Pydantic enables developers to define data models, also known as user-defined schemas.
- These models can include data types, validation rules, and default values.
- By establishing these models, developers can ensure that incoming data adheres to the expected structure and types.
- This validation process guarantees the reliability and integrity of the data used in your API.

Example: Basic Data Validation

- Let's say you want to validate user input for a blog post.
- You can define a Pydantic model to ensure the data is in the expected format.
- In this example, we define a BlogPost model with fields for the title and content.
- Pydantic automatically validates that title and content are strings.