European Energy Data AI Chatbot

Yuvan Gunuputi

September 9, 2025

1 Code Walkthrough and Explanation

This project is an AI-powered chatbot designed to provide real-time and forecast data on the European energy market. Users can ask questions in natural language about electricity load and generation forecasts for over 35 European countries, and the chatbot will retrieve and display the relevant statistics in a clean, user-friendly web interface. `Gradio`, and `entsoe-py` libraries.

1.1 Cell 1: Library Installation

The following commands were executed to ensure all necessary libraries are installed from the `requirements.txt` file and to upgrade `gradio` to its latest version.

```
%pip install -r requirements.txt
%pip install --upgrade gradio
```

1.2 Cell 2: Library Imports

This cell imports all the required libraries, organizing them by function for clarity.

```
import pandas as pd
import gradio as gr
from datetime import datetime, timedelta
import traceback
import warnings

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.prompts import PromptTemplate
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from typing import Optional

from entsoe import EntsoePandasClient
```

- **Data Handling:** `pandas` is essential for data manipulation and analysis, particularly for processing time-series data from the API.
- **User Interface:** `gradio` provides the framework for building a user-friendly, web-based interface for the chatbot.
- **AI Integration:** The `langchain` libraries are used to connect the application to the Google Gemini large language model, enabling natural language understanding.
- **API Interaction:** `entsoe.EntsoePandasClient` facilitates the retrieval of energy data from the ENTSO-E API.

1.3 Cell 3: API Key Loading

This cell demonstrates the secure loading of API keys from local files to avoid hardcoding sensitive credentials in the source code.

```
with open("gemini_api_key.txt", "r") as file:
    for line in file:
        if line.startswith("GEMINI_API_KEY"):
            gemini_api_key = line.split("=")[1].strip()
```

```
with open("entsoe_api_key.txt", "r") as file:
    for line in file:
         if line.startswith("ENTSOE_API_KEY"):
             entsoe_api_key = line.split("=")[1].strip()
```

1.4 Cell 4: Client Initialization

The primary API and AI clients are initialized in this cell using the keys loaded in the previous step.

```
model = ChatGoogleGenerativeAI(model="gemini-2.0-flash-lite", google_api_key=gemini_api_key,
↪  temperature=0)
client = EntsoePandasClient(api_key=entsoe_api_key)
```

1.5 Cell 5: Data Schema Definition

A `Pydantic BaseModel` is defined to create a structured schema for the AI's output, ensuring that the parsed data is consistent and reliable.

```
    class QuestionDetails(BaseModel):
        function_name: str = Field(description="The exact API function to call based on the user's query.")
        country: str = Field(description="The full English name of the country.")
        start_date: str = Field(description="The start date for the query in 'YYYY-MM-DD' format.")
        end_date: str = Field(description="The end date for the query in 'YYYY-MM-DD' format.")

        psr_type: Optional[str] = Field(None, description="Power Source Type (e.g., 'solar', 'wind',
        ↪  'nuclear').")
        granularity: Optional[str] = Field(None, description="The level of detail: 'plant' or 'unit'.")
        type_marketagreement_type: Optional[str] = Field(None, description="Market agreement type for
        ↪  contracted reserves (e.g., 'A01').")
```

1.6 Cell 6: Output Parser

The `PydanticOutputParser` is configured to handle the conversion of the language model's text-based JSON output into a valid `QuestionDetails` object.

```
parser = PydanticOutputParser(pydantic_object=QuestionDetails)
```

1.7 Cell 7: Prompt Engineering

The core instructions for the language model are defined in this prompt string, including rules for parsing, function mapping, and examples of desired output.

```
prompt_string = """
You are an expert API assistant for the ENTSO-E energy platform. Your job is to parse a user's query and
↪  convert it into a structured JSON object.

Today's date is {current_date}.

--- FUNCTION MAPPING ---
Use this table to determine the correct 'function_name'.

| If the user asks for...              | Use this 'function_name'      |
|--------------------------------------|-------------------------------|
| Actual load, consumption, or demand | query_load                    |
| Generation forecast or prediction   | query_generation_forecast     |

--- PARAMETER EXTRACTION RULES ---
1.  **Country**: Extract the full English country name from the user's query.
2.  **Date Range**:
    - Based on the query and today's date ({current_date}), figure out the correct `start_date` and
    ↪  `end_date` in 'YYYY-MM-DD' format.
    - If the user says "yesterday", use yesterday's date.
    - If no date is mentioned, assume the user means "today".

--- EXAMPLES ---
- Query: "What is the current load in France?"
  Expected Output: {{ "function_name": "query_load", "country": "France", "start_date": "2025-09-08",
  ↪  "end_date": "2025-09-08" }}

- Query: "Show me the generation forecast for Germany today"
```

```
  Expected Output: {{ "function_name": "query_generation_forecast", "country": "Germany", "start_date":
  ↪  "2025-09-08", "end_date": "2025-09-08" }}

- Query: "What was the load in Spain yesterday?"
  Expected Output: {{ "function_name": "query_load", "country": "Spain", "start_date": "2025-09-07",
  ↪  "end_date": "2025-09-07" }}

{format_instructions}

User Query:
{query}
"""
```

1.8 Cell 8: Prompt Template

This template combines the static prompt with dynamic inputs and the parser's formatting instructions to create a complete message for the LLM.

```
prompt = PromptTemplate(
    template=prompt_string,
    input_variables=["query", "current_date"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)
```

1.9 Cell 9: The LangChain Expression Language (LCEL) Chain

This one-line of code establishes the entire workflow, from input to final output, in a single, readable pipeline.

```
chain = prompt | model | parser
```

1.10 Cell 10: Data Mappings

These dictionaries provide the necessary mappings to convert human-readable names into the API-specific codes required for data retrieval.

```
EUROPEAN_COUNTRY_CODES = {
    "albania": "AL", "austria": "AT", "belgium": "BE", "bosnia and herzegovina": "BA",
    "bulgaria": "BG", "croatia": "HR", "cyprus": "CY", "czech republic": "CZ", "denmark": "DK",
    "estonia": "EE", "finland": "FI", "france": "FR", "georgia": "GE", "germany": "DE",
    "greece": "GR", "hungary": "HU", "iceland": "IS", "ireland": "IE", "italy": "IT",
    "kosovo": "XK", "latvia": "LV", "lithuania": "LT", "luxembourg": "LU", "malta": "MT",
    "moldova": "MD", "montenegro": "ME", "netherlands": "NL", "north macedonia": "MK",
    "norway": "NO", "poland": "PL", "portugal": "PT", "romania": "RO", "serbia": "RS",
    "slovakia": "SK", "slovenia": "SI", "spain": "ES", "sweden": "SE", "switzerland": "CH",
    "turkey": "TR", "ukraine": "UA", "united kingdom": "GB",
}
```

1.11 Cell 11: API Dispatcher

The `call_entsoe_api_dispatcher` function acts as an intermediary, directing the structured AI output to the correct `entsoe-py` API call.

```
def call_entsoe_api_dispatcher(client, question: QuestionDetails):
    start = pd.Timestamp(question.start_date, tz='Europe/Brussels')
    end = pd.Timestamp(question.end_date, tz='Europe/Brussels') + timedelta(days=1)

    country_code = EUROPEAN_COUNTRY_CODES.get(question.country.lower()) if question.country else None

    if not country_code:
        return "Query is missing a valid country."

    func_name = question.function_name

    try:
        if func_name == 'query_load':
            return client.query_load(country_code, start=start, end=end)

        elif func_name == 'query_generation_forecast':
            return client.query_generation_forecast(country_code, start=start, end=end)
```

```
        else:
            return f"Function '{func_name}' is not supported by this dispatcher."

    except Exception as e:
        return f"API call failed for function '{func_name}' with error: {e}"
```

1.12 Cell 12: Data Formatting

This function processes the raw API data, calculates summary statistics, and formats it into a user-friendly table for display.

```
def format_data_as_table(data, details: QuestionDetails):
    title = details.function_name.replace('query_', '').replace('_', ' ').title()

    if isinstance(data, str):
        return data, None

    if data.empty:
        return f"No data found for {title} in {details.country.capitalize()}", None

    try:
        stats_series = data.iloc[:, 0] if isinstance(data, pd.DataFrame) else data

        summary = (
            f"**Summary for {title} in {details.country.capitalize()}**\n"
            f"**Period:** {details.start_date} to {details.end_date}\n\n"
            f"Mean: **{stats_series.mean():.2f}**\n"
            f"Min: **{stats_series.min():.2f}**\n"
            f"Max: **{stats_series.max():.2f}**"
        )
    except Exception as e:
        summary = f"**Data for {title} in {details.country.capitalize()}**\n(Could not calculate summary
        ↪  statistics due to an error.)"

    unit = "EUR/MWh" if 'price' in details.function_name.lower() else "MW"
    table_df = pd.DataFrame(data).reset_index()

    if len(table_df.columns) >= 2:
        table_df.columns = ['Timestamp'] + [f'Value_{i}' for i in range(1, len(table_df.columns))]
        table_df.rename(columns={'Value_1': f'Value ({unit})'}, inplace=True)

    table_df['Timestamp'] = pd.to_datetime(table_df['Timestamp']).dt.strftime('%Y-%m-%d %H:%M')

    return summary, table_df
```

1.13 Cell 13: Main Query Processor

The `process_query` function orchestrates the entire workflow, from parsing the user's input to fetching and formatting the final output.

```
def process_query(question):
    if not question:
        return "Please ask a question.", None
    try:
        current_date_str = datetime.now().strftime('%Y-%m-%d')
        parsed_details = chain.invoke({"query": question, "current_date": current_date_str})

        data = call_entsoe_api_dispatcher(client, parsed_details)

        summary, table = format_data_as_table(data, parsed_details)
        return summary, table
    except Exception as e:
        traceback.print_exc()
        return f"Sorry, a critical error occurred. Error: {e}", None
```

1.14 Cell 14 & 15: Gradio Interface

These cells define and launch the `Gradio` web interface, which serves as the front-end for the chatbot.

```
interface = gr.Interface(
    fn=process_query,
    inputs=gr.Textbox(lines=2, placeholder="e.g., Show me electricity prices in Germany yesterday..."),
```

```
    outputs=[
        gr.Markdown(label="Summary"),
        gr.DataFrame(label="Data Table", wrap=True)
    ],
    title=" European Energy Data Chatbot (Table View)",
    description="Ask me about day-ahead electricity prices or total load for various European countries.",
    flagging_mode="never"
)

interface.launch()
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=ResourceWarning, message="unclosed <socket.socket.*>")
```