**This project is a Python-based text analysis pipeline designed to process cryptocurrency-related articles, extract meaningful insights, and provide answers to user queries. The pipeline begins with acquiring data from web articles, converting them into PDFs, extracting text, preprocessing it, vectorizing it for analysis, performing sentiment analysis, and retrieving ranked answers using the BM25 algorithm. The codebase is modular, with contributions from multiple authors, and leverages a variety of NLP and machine learning libraries.**

**OBJECTIVES:**

**- Scrape and gather cryptocurrency-related articles from the web.**

**- Extract and refine text from these articles for analysis.**

**- Analyze sentiment and identify cryptocurrency-specific entities.**

**- Deliver ranked, relevant answers to user queries based on the processed text.**

**DATA ACQUISITION**:

This step scrapes article URLs from the cryptocurrency news website, thenewscrypto.com for five specified coins: Dogecoin, Bitcoin, Ethereum, Solana, and Hamster. The script fetches up to 10 article links per coin from the first page of search results and saves them into individual CSV files (e.g., dogecoin\_news\_urls.csv). It incorporates error handling, logging, and a user-agent header to mimic a browser request, ensuring robust and reliable web scraping.

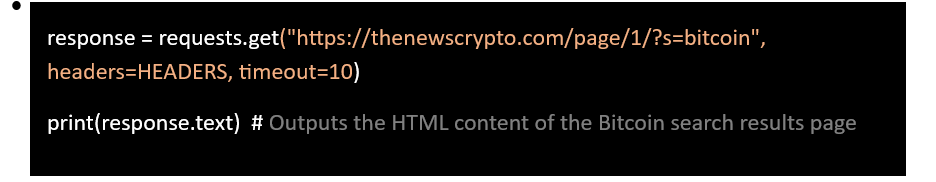
**Key Features**

* Scrapes only the first page of search results per coin.
* Limits extraction to 10 articles per page to maintain consistency.
* Uses a user-agent header to avoid being blocked by the website.
* Implements logging to track progress and errors.
* Saves results in CSV format with coin names and URLs.

**Libraries used:**

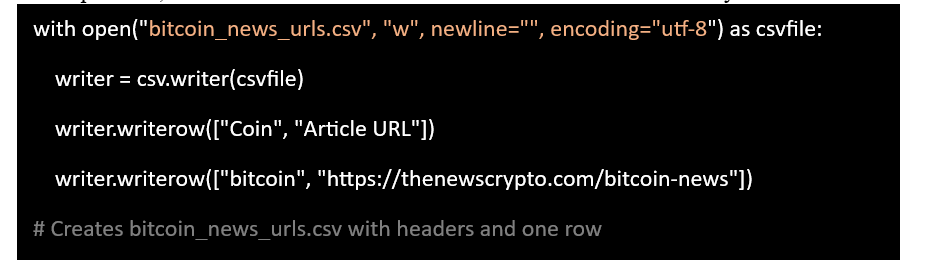
1. **Requests**: https://pypi.org/project/requests/

* It is used for sending the HTTP request to the website and then retrieving the HTML content of the search results.
* Used in the ***get\_article\_link()*** function to fetch the webpage content for a given coin and page number via ***requests.get(url, headers= HEADERS, timeout=10)***
* **Efficiency**: requests simplify HTTP interactions, abstracting away low-level socket programming and connection handling. This makes it highly efficient for fetching web pages compared to manual alternatives like **urllib.**
* **Reliability**: The library handles redirects, connection errors, and timeouts gracefully. The timeout=10 parameter ensures the script doesn’t hang indefinitely, while the try-except block catches ***RequestException*** errors (e.g., network failures).
* **Flexibility**: The ability to pass custom headers (e.g., ***HEADERS***) is crucial for avoiding anti-scraping measures like IP blocking, as many websites reject requests without a valid user-agent.
* Without requests, the script would require complex manual HTTP request construction, reducing readability and increasing the likelihood of errors. Its use ensures reliable page retrieval, critical for subsequent parsing.

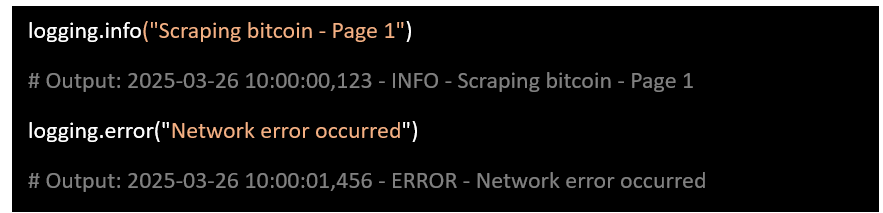


1. **CSV**: https://docs.python.org/3/library/csv.html

* For providing functionality to read and write CSV (Comma-Separated Values) files.
* Used in the ***scrape\_articles\_for\_coin()*** function to save scraped article URLs into a CSV file (e.g., dogecoin\_news\_urls.csv) with columns "Coin" and "Article URL" via ***csv.writer***.
  + **CSV Writer**: Creates a writer object to write rows to the CSV file.
  + **newline=""**: Ensures consistent line endings across platforms (e.g., Windows, Linux).
  + **Encoding**: Uses utf-8 to support special characters in URLs or coin names.
* **Simplicity**: csv provides a straightforward interface for writing tabular data, eliminating the need to manually format comma-separated strings. This reduces errors like missing delimiters or improper escaping.
* **Portability**: The library ensures the output CSV files are compatible with standard tools (e.g., Excel, pandas), making them reusable for further processing.
* **Robustness**: Handles edge cases like special characters or quotes in URLs automatically, thanks to utf-8 encoding and built-in escaping mechanisms.
* **Context in Script**: Without csv, the script would need to manually write comma-separated lines (e.g., f.write(f"{coin},{link}\n")), increasing complexity and error risk. Its use ensures clean, structured output files, critical for downstream tasks like PDF conversion or analysis

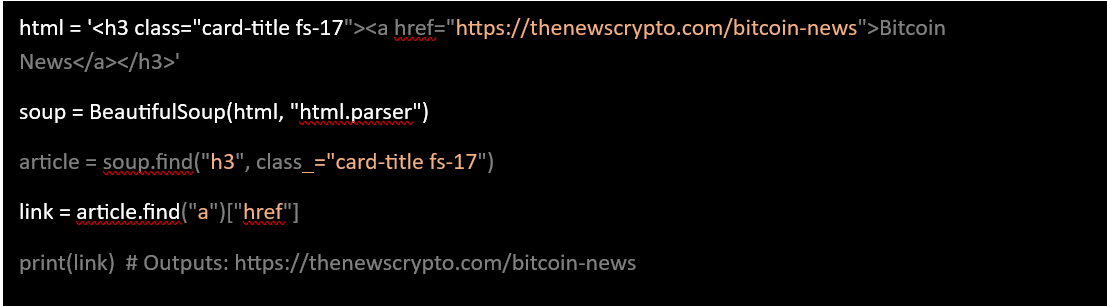
****

1. logging: <https://docs.python.org/3/library/logging.html>

* Enables logging of events (info, warnings, errors) during execution for debugging and monitoring.
* Log messages like "Scraping bitcoin - Page 1" or errors if the request fails.
* Configured at the start with ***logging.basicConfig()*** and used throughout to log scraping progress (e.g., ***logging.info***), warnings (e.g., ***logging.warning***), and errors (e.g., ***logging.error***).
  + **Level Setting**: Sets the logging level to INFO to capture all events.
  + **Format**: Customizes log messages with timestamp, level, and message ***(%(asctime)s - %(levelname)s - %(message)s).***
  + **Log Types**: Uses info for progress, warning for non-critical issues, and error for exceptions.
* **Debugging**: Provides a detailed trace of the scraping process (e.g., "Scraping bitcoin - Page 1"), making it easy to identify where failures occur (e.g., HTTP errors).
* **Monitoring**: Logs the number of articles found per page, helping verify the script’s success without manual inspection of output files.
* **Robustness**: Captures exceptions (e.g., network timeouts) with ***logging.error***, ensuring issues are documented rather than silently failing.
* Without ***logging***, debugging would rely on print statements, which lack timestamps and severity levels, making it harder to track issues in a long-running process. Its use enhances maintainability and reliability, especially for scraping multiple coins.
* **Example**

1. Bs4 [beautiful Soup]: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

* Used for scraping websites. And parsing the HTML content
* **Ease of Use**: ***BeautifulSoup*** simplifies HTML parsing with an intuitive API, allowing quick extraction of article links without writing complex regular expressions or DOM traversal logic.
* **Flexibility**: Handles malformed HTML gracefully, which is common on real-world websites, ensuring the script doesn’t break if the page structure varies slightly.
* **Precision**: The use of ***class\_="card-title fs-17***" targets specific article titles, reducing the risk of scraping irrelevant links (e.g., ads or navigation).
* Without ***bs4***, parsing HTML would require manual string manipulation or a less robust library like ***re***, increasing complexity and fragility. Its effectiveness lies in reliably extracting structured data from unstructured HTML, making it indispensable for web scraping.



1. **Text Extraction:**
2. trafilatura: https://trafilatura.readthedocs.io/en/latest/

A web scraping and content extraction library designed to fetch and extract clean, readable text from HTML pages, removing boilerplate (e.g., ads, navigation). Unlike general-purpose scraping tools like BeautifulSoup, trafilatura specializes in extracting the main article content efficiently, which is ideal for news articles.

* Used in ***process\_coin()*** to download article pages with ***trafilatura.fetch\_url(url)*** and extract their main content with ***trafilatura.extract(downloaded)***.
* **Key Features Utilized**:
  + **URL Fetching**: Downloads HTML content from a given URL.
  + **Content Extraction**: Strips away non-essential elements to isolate the main article text.

**Effectiveness**

* **Specialization**: Unlike general-purpose scraping tools like BeautifulSoup, trafilatura is optimized for extracting main article content, making it highly effective for news articles where boilerplate removal is critical. It uses heuristics and machine learning to identify primary content, reducing manual parsing effort.
* **Robustness**: Handles diverse webpage structures and gracefully returns None if downloading or extraction fails, allowing the script to continue without crashing. This is evident in the checks for downloaded and content.
* **Efficiency**: Combines fetching and extraction in a streamlined process, minimizing the need for separate HTTP requests (e.g., via requests) and parsing steps. This reduces latency compared to a requests + BeautifulSoup approach.
* **Context in Script**: Without trafilatura, the script would need to fetch pages with requests and manually parse HTML with BeautifulSoup, requiring custom logic to filter boilerplate. Its use ensures clean, relevant text extraction, crucial for generating meaningful PDFs for downstream analysis.

1. pdfkit: https://pypi.org/project/pdfkit/

Converts HTML files to PDF format using the wkhtmltopdf command-line tool (requires wkhtmltopdf to be installed separately).

D**ATA PREPROCESSING**

This step cleans, tokenizes, stems, and lemmatizes the text while preserving cryptocurrency-specific terms, preparing it for downstream tasks like vectorization or sentiment analysis. It processes the text sentence-by-sentence, provides progress updates, and saves the cleaned output to cleaned\_data.txt.

Key Features:

* Splits text into sentences using a simple newline split for efficiency.
* Applies comprehensive cleaning: lowercase conversion, special character removal, tokenization, stemming, and lemmatization.
* Uses a progress bar to monitor processing of large datasets.
* Preserves a predefined set of cryptocurrency terms to maintain domain-specific context.

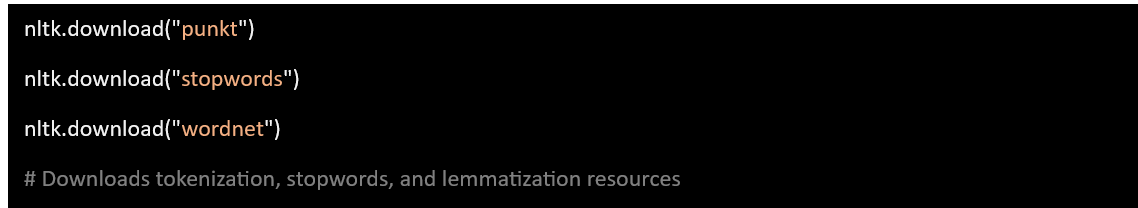
Libraries Used:

1. nltk (Natural Language Toolkit): https://www.nltk.org/

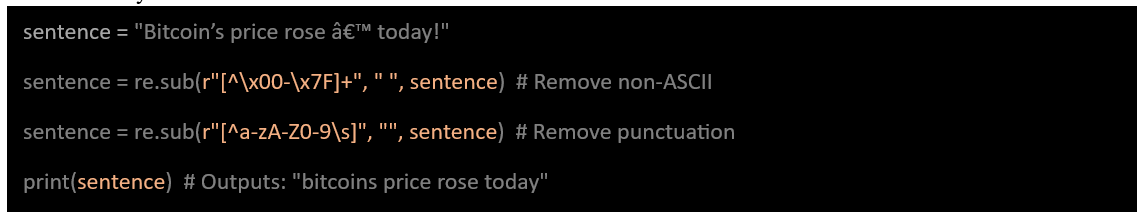
This library is used for tasks like tokenization, stopword removal, and lemmatization. It provides pre-built tools and resources (e.g., tokenizers, stopwords, lemmatizers) that simplify text preprocessing, avoiding the need to write these from scratch.

* Provides a broad suite of natural language processing tools and resources, serving as the backbone for tokenization, stopwords, and stemming/lemmatization in this script.
* Used indirectly through its submodules (***nltk.tokenize, nltk.corpus, nltk.stem***) and to download required resources (***punkt, stopwords, wordnet***) via ***nltk.download(***).
  + **Resource Downloads**: Ensures necessary datasets are available for tokenization and lemmatization.
  + **Foundation**: Acts as the parent library for specialized NLP tasks.
* **Versatility**: ***nltk*** is a comprehensive NLP library, offering pre-built tools that save time compared to custom implementations. Its modular design allows specific submodules to handle distinct tasks.
* **Reliability**: The downloaded resources (***punkt, stopwords, wordnet***) are well-tested and widely used, ensuring consistent preprocessing results.
* Without nltk, the script would require manual tokenization, stopword lists, and lemmatization logic, significantly increasing complexity. Its role is foundational, enabling the use of specialized submodules for efficient text processing.

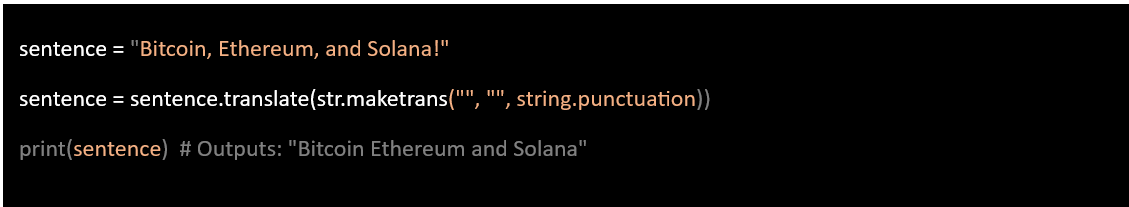
**Example**



1. **re:** https://docs.python.org/3/library/re.html

* Provides regular expression matching operations to remove unwanted characters from text.
* Used in ***preprocess\_sentence()*** to remove non-ASCII characters and punctuation via ***re.sub().***
  + **Pattern Matching**: Removes specific character sets (e.g., non-ASCII, non-alphanumeric) using regex patterns.
  + **Substitution**: Replaces matched patterns with spaces or empty strings.
* **Precision**: re allows fine-grained control over text cleaning, effectively targeting non-ASCII characters ***([^\x00-\x7F])*** and punctuation ***([^a-zA-Z0-9\s]).***
* **Speed**: Regular expressions are optimized for string operations, making them faster than iterative character checks for large texts.
* Without re, cleaning special characters would require slower, less flexible methods (e.g., multiple ***replace()*** calls). Its use ensures robust text normalization, critical for consistent tokenization and analysis.

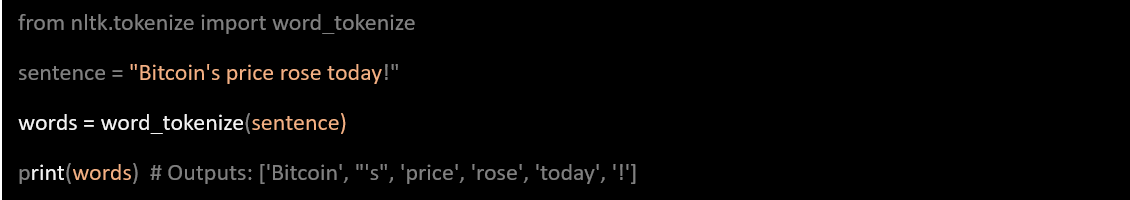
1. tring: <https://docs.python.org/3/library/string.html>

* Supplies string manipulation utilities, specifically for punctuation removal.
* **Usage in Code**: Used in preprocess.***sentence***() to create a translation table for removing punctuation via ***str.maketrans("", "", string.punctuation).***
  + **Punctuation Constants**: Provides ***string.punctuation***, a predefined set of punctuation characters ***(e.g., !"#$%&'()\*+,-./:;<=>?@[\]^\_{|}~`).***
  + **Translation Table**: Generates a mapping to strip punctuation efficiently.
* **Simplicity**: ***string.punctuation*** offers a ready-made list of characters to remove, avoiding the need to define them manually.
* **Efficiency**: The ***translate()*** method with a translation table is faster than ***regex*** or iterative replacements for punctuation removal.
* While re could handle punctuation removal, string provides a cleaner, more efficient alternative for this specific task. Its use complements ***re*** by focusing on punctuation, enhancing the overall cleaning process.

**4. nltk.tokenize: https://www.nltk.org/api/nltk.tokenize.html**

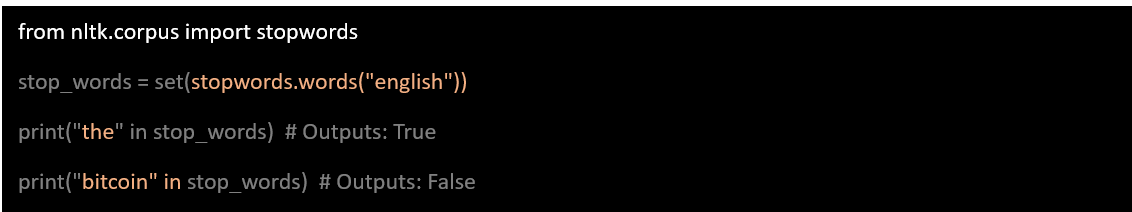
**Role**

* **Purpose**: Provides tokenization tools to split text into words or sentences.
* Used in preprocess\_***sentence()*** to tokenize a sentence into words via ***word\_tokenize(sentence)***.
  + **Word Tokenization**: Splits a sentence into individual words based on the punkt tokenizer.
* **Accuracy**: ***word\_tokenize*** leverages the punkt model, trained on diverse English texts, to accurately handle punctuation and contractions (e.g., "Bitcoin's" → ["Bitcoin", "'s"]).
* **Convenience**: Eliminates the need for custom tokenization logic, which would be error-prone and less robust (e.g., simple ***split()*** misses edge cases).
* Without ***nltk.tokenize***, the script might use ***split(),*** which fails on complex punctuation or multi-word crypto terms. Its effectiveness ensures precise word-level processing, critical for stemming and lemmatization.

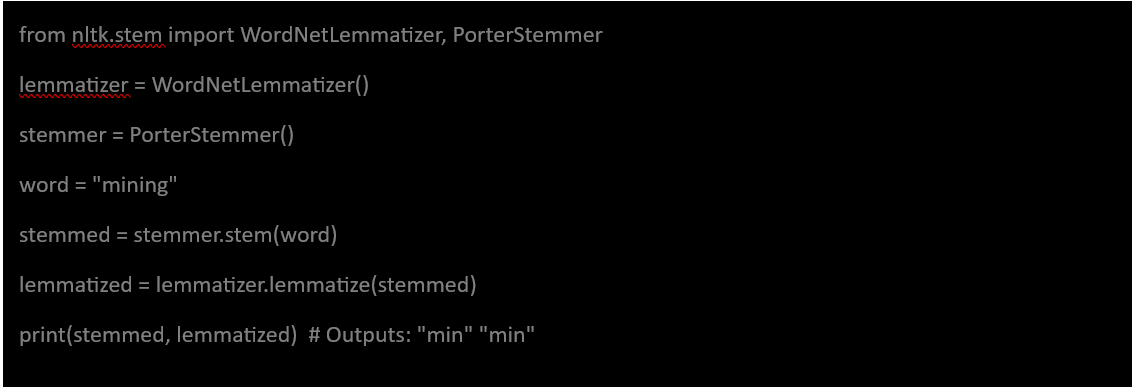


[**nltk.corpus**](https://www.nltk.org/api/nltk.corpus.html)

* Provides access to linguistic resources, such as a list of stopwords.
* Used to load English stopwords into a set via ***stopwords.words("english"***) in the setup.
  + **Stopwords**: Supplies a precompiled list of common English words (e.g., "the", "is", "and") to filter out.
* **Efficiency**: Loading stopwords into a set enables fast lookup during processing, crucial for large datasets.
* **Standardization**: Uses a widely accepted stopword list, ensuring consistency with NLP best practices.
* Without ***nltk.corpus,*** the script would require a custom stopword list, risking omissions or inconsistencies. Its use reduces noise in the cleaned text, improving the quality of downstream analysis.

**5. nltk.stem : https://www.nltk.org/api/nltk.stem.html**

* **Purpose**: Provides stemming and lemmatization tools to reduce words to their base forms.
* **Usage in Code**: Used in preprocess\_***sentence()*** to initialize and apply WordNetLemmatizer and ***PorterStemmer*** via ***stemmer.stem(word)*** and ***lemmatizer.lemmatize(stemmed\_word).***
* **Key Features Utilized**:
  + **PorterStemmer**: Applies rule-based stemming (e.g., "running" → "run").
  + **WordNetLemmatizer**: Uses WordNet to lemmatize words to their dictionary form (e.g., "better" → "good").
* **Normalization**: Combining stemming and lemmatization ensures words are reduced to consistent base forms, improving text similarity for analysis (e.g., "mining" and "mined" → "mine").
* **Precision**: ***WordNetLemmatizer*** leverages a lexical database for accurate base forms, while ***PorterStemmer*** handles simpler cases efficiently.
* Without ***nltk.stem***, words like "prices" and "pricing" would remain distinct, reducing the effectiveness of vectorization or retrieval. Its use enhances text consistency while preserving crypto terms.



S**entiment Analysis using Blob**

This step performs sentiment analysis on the sentences prepared, adding sentiment scores and categories (positive, negative, neutral) to the DataFrame. It then visualizes the distribution of sentiments across all sentences, providing insights into the tone of the crypto news articles.

The libraries used:

1. **textBlob: https://textblob.readthedocs.io/en/dev/**

A simple NLP library that provides tools for tasks like sentiment analysis, part-of-speech tagging, and text processing.

It offers an easy-to-use interface for sentiment analysis, calculating polarity (positive/negative) and subjectivity (objective/subjective) without requiring complex model training.

Analyzes the sentiment of raw sentences using **TextBlob(x) sentiment, polarity,** and subjectivity**.**

Used in ***analyze\_crypto\_sentiment()*** to create a ***TextBlob*** object and compute polarity via ***blob.sentiment.polarity.***

 **Simplicity**: Requires no model training, making it ideal for quick sentiment analysis compared to complex libraries like transformers.

 **Accuracy**: Uses a pre-trained pattern-based analyzer, effective for general English but may miss nuanced crypto-specific sentiment (e.g., "mining" as positive or negative).

 **Context in Script**: Without ***textblob,*** sentiment analysis would require a custom model or external API, increasing complexity. Its ease of use makes it effective for rapid prototyping and visualization.

*Example: TextBlob("Dogecoin prices rose today!").sentiment returns:*

* + *polarity: ~0.25 (positive, as "rose" suggests growth).*
  + *subjectivity: ~0.2 (fairly objective).*

*Category: positive (since 0.25 > 0.1).*

1. **matplotlib.pyplot**: https://matplotlib.org/3.5.3/api/\_as\_gen/matplotlib.pyplot.html
   * **Purpose**: A plotting library for creating visualizations like bar charts, line graphs, etc. Used to plot sentiment distribution with ***plt.bar()*** in the main execution.

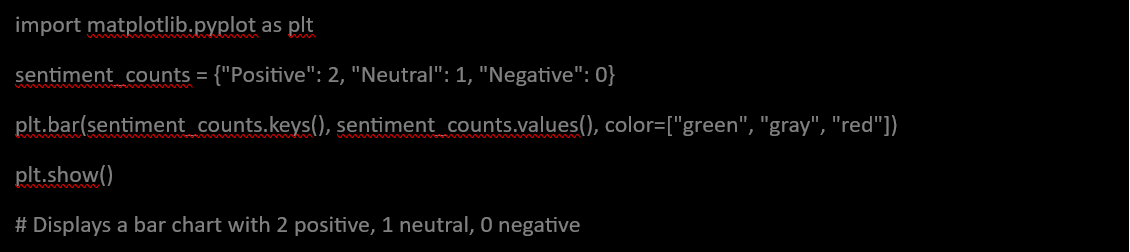
It’s widely used for quick, customizable data visualization, ideal for showing sentiment distribution.

Generates a bar chart to display the count of sentences in each sentiment category.

Visualizations: Counts all categories (e.g., 50 positive, 30 neutral, 20 negative) and plots them in a bar chart.

* + **Visualization Example**:
    - Bar chart with:
      * Green bar (positive)
      * Gray bar (neutral)
      * Red bar (negative)
  + Without ***matplotlib***, sentiment distribution would require manual counting or external software, reducing interactivity. Its use enhances insight generation from sentiment data.

**Example:**



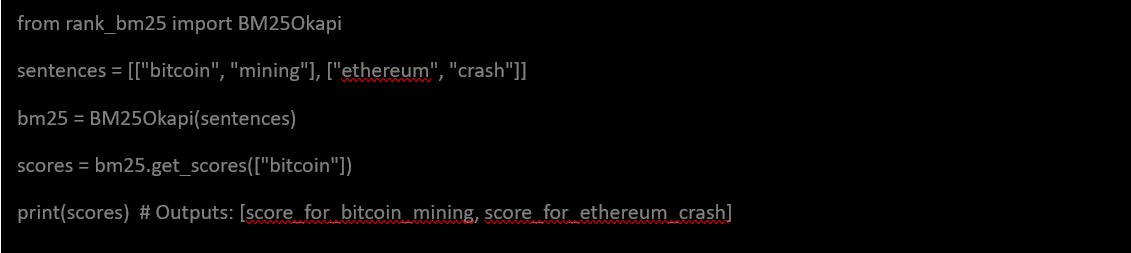
**Implementation of BM25**

This step creates a searchable index of preprocessed sentences using the BM25 algorithm, processes user queries, and retrieves the most relevant sentences as answers. It integrates sentiment data from Step 5 and provides a ranked list of responses with metadata.

Libraries used:

1. **rank\_bm25**: https://pypi.org/project/rank-bm25/
   * Implements the BM25 (Best Matching 25) algorithm, a ranking function used in information retrieval to score document relevance based on query terms.
   * BM25 is efficient and effective for text search, balancing term frequency and document length, making it ideal for retrieving relevant sentences without needing a complex machine learning model.
     + B**M25Okapi:** Creates an index from tokenized sentences and scores them against queries. Used in ***CryptoAnswerRetrieval*** to initialize a ***BM25Okapi*** index with tokenized sentences and score them against a query via ***self.bm25 = BM25Okapi()*** and ***bm25.get\_scores().***

The process followed:

1. **Creating the BM25 Index (Function: create\_bm25\_index)**:
   * Convert preprocessed sentences into a tokenized corpus.
   * Initialize a ***BM25Okapi index*** with the corpus.
2. **Query Preprocessing (Function: preprocess\_query)**:
   * Apply the same preprocessing to the user’s query to ensure consistency with the indexed data.
3. **Searching with BM25 (Function: search\_with\_bm25)**:
   * Preprocess the query and tokenize it into words.
   * Compute relevance scores for all sentences using the BM25 index.
   * Add scores to a copy of the input DataFrame, sort by score, and filter to the top N results.
   * Warn if the highest score is below a threshold (0.05), indicating poor matches.
   * Return the top N results.
4. **Execution**:
   * Create the BM25 index from ***sentiment\_df.***
   * Test with an example query ("What is the latest trend in Bitcoin price?").
   * Display the top 3 results with source, score, sentiment, and raw sentence.

**Integration of NLP Model with Flask Backend**

This section documents the transition of the NLP model from a CLI-based interface to a web-integrated backend architecture using Flask. The modifications allow the entire model pipeline to function via HTTP requests, improving accessibility, automation, and frontend interaction.  
The model.py file is adapted for integration with the Flask backend (app.py) instead of using a command-line interface. The process\_query() function is designed to take structured inputs (coin, query) and return structured JSON outputs for rendering in the frontend.

The function `process\_query(coin, query)` is the main bridge between the Flask backend and the NLP model. It takes a cryptocurrency name and user input as arguments and returns structured results suitable for JSON serialization and frontend display.

**This function:**  
- Analyzes sentiment using TextBlob  
- Retrieves top matching answers using BM25 and TF-IDF algorithms  
- Combines results and sorts them  
- Returns a dictionary with formatted output and raw data for logging  
  
**Key Return Structure:**  
{  
 "query": query,  
 "coin": coin,  
 "sentiment": "Positive (Score: 0.65)",  
 "top\_answers": [  
 "1. Bitcoin is a decentralized cryptocurrency... (Score: 6.53)"  
 ],  
 "raw\_top\_results": [(sentence, score)]  
}  
The `raw\_top\_results` field is retained only for server-side logging purposes. It is removed before the frontend receives the data. This keeps logs comprehensive while maintaining frontend efficiency.

**Query Logging via logger.py**

Each processed query is logged into `query\_logs.json` for tracking. The log entry includes:  
- Timestamp of query  
- Coin name  
- User question  
- Detected sentiment  
- Top answer text and scores  
This is handled by the `log\_query()` function, which appends logs in a JSON array format using Python's built-in `json` module.

**Summing up Key Enhancements:**

**1. Unified Query Handler for API**

def process\_query(coin, query):

* Replaces CLI interaction with a reusable backend-compatible function.
* Accepts two parameters: the selected cryptocurrency (coin) and user question (query).
* Returns a structured response dictionary that includes:
  + Original query
  + Coin name
  + Sentiment result (with score)
  + Ranked top 3 answers (as formatted strings)
  + Raw answer-score pairs for logging

**Why this matters:**  
Previously, the output was printed to the terminal. Now, it is returned to the Flask route /query where it can be serialized to JSON and sent to the frontend.

**2. Output Formatting for Frontend Rendering**

"top\_answers": [

f"{i+1}. {answer[:500]}... (Score: {score:.2f})"

for i, (answer, score) in enumerate(top\_results)

]

* Ensures each result is trimmed and human-readable.
* Designed for HTML rendering inside the web interface (index.html).

**3. Compatibility with Logging Module**

"raw\_top\_results": top\_results

* This line is included in the returned dictionary **only for backend logging**.
* It is removed before sending the response to the frontend in app.py.

**4. Coin Validation Integration**

COINS = ["bitcoin", "ethereum", "solana", "dogecoin", "hamstercoin", "cardano", "general crypto"]

* Now shared across model.py and app.py.
* Ensures any coin selected in the frontend is validated before query processing.

**5. Removed Command-Line Interface**

* The interactive loop when writing code for model (asking the user to type the coin and query) has been removed.
* Replaced with stateless processing suited for HTTP requests.

**# query\_logs.json**

**Purpose:** It is used to store a history of user queries and system responses during the text analysis process.

Usage in code:

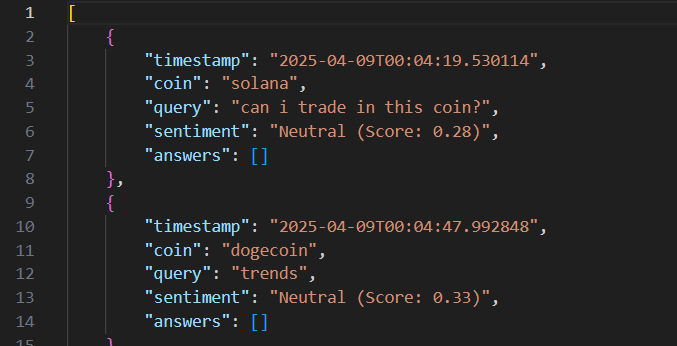
* **Error Tracking:** Can log query failures (e.g., empty results or low BM25 scores).
* **Audit Trail:** Used to validate if the system is returning accurate and relevant answers over time.

Key Features:

* **JSON Format:** Stores logs in a structured JSON array for easy access and readability.
* **Log Entries:** Each log typically contains a timestamp, query text, response, sentiment, and confidence score.

Importance:

* **Transparency:** Enables retrospective review of system outputs.
* **Model Improvement:** Helps analyze mismatches between query intent and retrieved sentences.
* **User Insight:** Useful for visualizing user interest trends over time.
* **Testing & Debugging:** Makes regression testing easier by replaying previous queries.



**# Keywords.py**

**Purpose:** It is responsible for extracting or managing key cryptocurrency-related terms or phrases from articles. These keywords are used to enhance search relevance, preserve domain-specific context during preprocessing, and improve user query matching in the BM25 retrieval step.

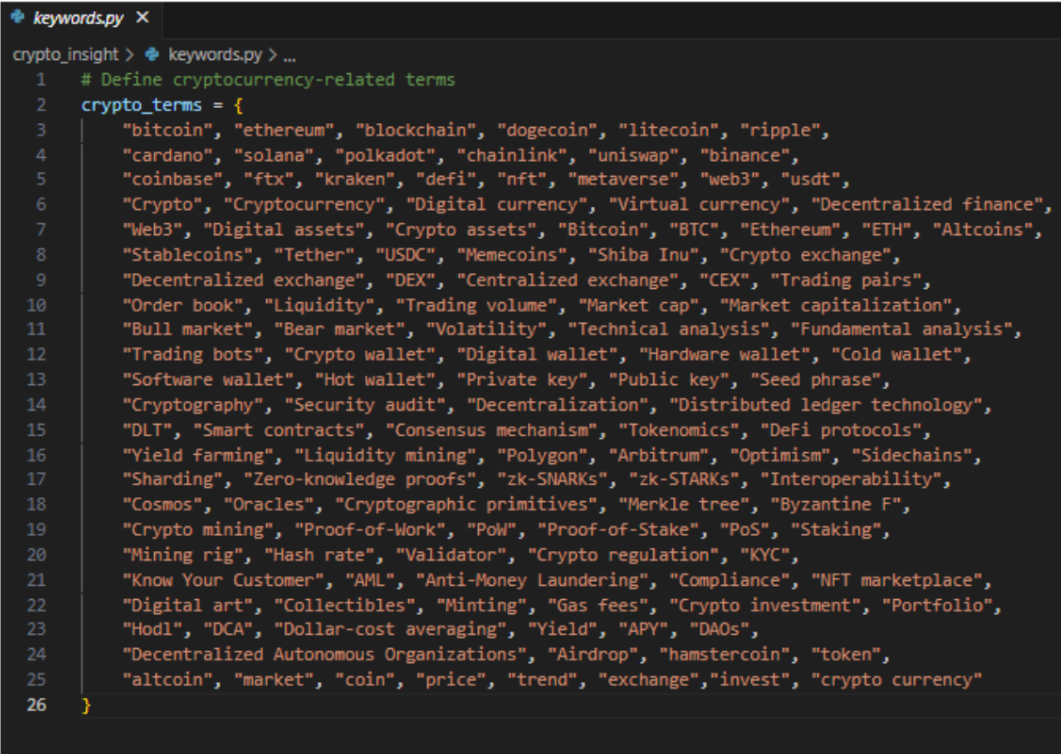
**Key Features:**

* **Crypto-Specific Vocabulary:** Maintains a curated list of crypto terms (e.g., "blockchain", "DeFi", "halving", "altcoin").
* **Custom Stopword Exceptions:** Prevents removal of important crypto terms during preprocessing (e.g., during stopword filtering).
* **Keyword Matching:** May provide utility functions to identify or highlight keyword presence in text.
* **Query Expansion (if implemented):** Can expand user queries with related keywords for broader match.

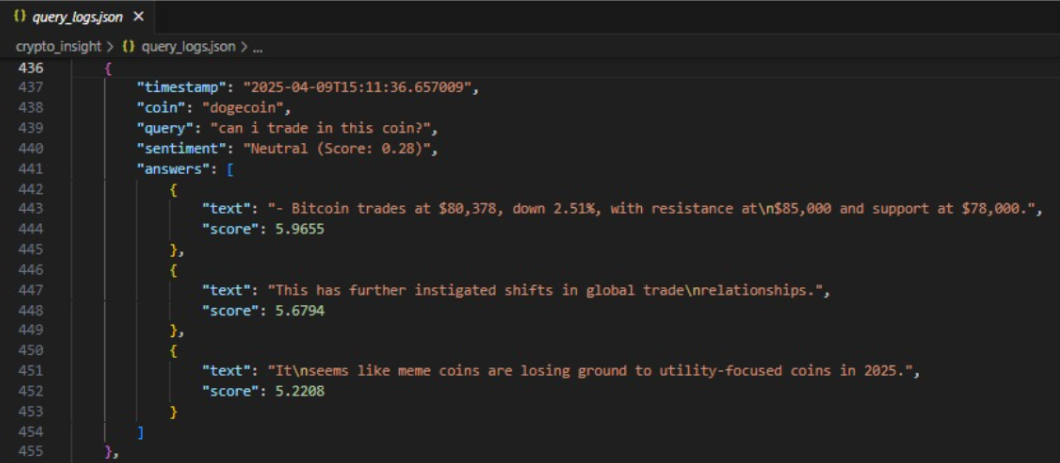
**Usage in code:**

* **Preprocessing (Step: preprocess\_sentence)**
* Ensures that key crypto terms are **not stemmed or lemmatized** to preserve their semantic identity.
* Prevents accidental removal during stopword or special character cleaning.
* **Sentiment or Relevance Analysis:**
* Used to **filter or tag** sentences that contain crypto-related terms for priority scoring.
* **BM25 Index Creation:**
* Can be used to label which sentences are “keyword-rich” to weigh them higher in ranking.

**Importance:**

* **Domain Sensitivity:** Crypto-specific terms often don't behave like general English (e.g., “hodl” isn’t in any dictionary). Preserving these terms improves model accuracy.
* **Search Optimization:** Ensures BM25 index or query expansion functions operate with **context-aware terms**.
* **User Intent Matching:** Improves alignment between user queries and text content by anchoring on domain keywords.

**Result of keywords.py in a separate json file for future QA:**



**# Logger.py**

**Purpose:** It is used to configure and manage logging across the entire crypto text analysis pipeline. It centralizes the setup of the logging system so that other modules can easily import and use a consistent logging format and level. This helps in monitoring, debugging, and auditing the system.

**Key Features:**

* **Centralized Logger Configuration:** Defines logging settings (level, format, output file).
* **Reusable Across Modules:** Any script (e.g., scraper.py, bm25.py) can import the configured logger and write logs.
* **File and Console Output:** Optionally writes logs to both the terminal and a log file (e.g., pipeline.log).
* **Custom Format:** Includes timestamps, log levels (INFO, WARNING, ERROR), and message content.

**Usage in code:**

* Imported into modules like scraper.py or bm25.py to log progress updates (e.g., "Scraping Bitcoin - Page 1"), warnings (e.g., missing articles), and errors (e.g., failed network requests) using logger.info(), logger.warning(), and logger.error().
* Ensures all log messages are consistently formatted and saved to both the console and a log file (pipeline.log), supporting easy debugging and execution tracking.

**Importance:**

* **Traceability:** Tracks every step (e.g., "Preprocessing complete", "BM25 index created").
* **Debugging:** Helps locate where and why failures occurred (e.g., network issues, parsing errors).
* **Monitoring Progress:** Useful for long-running processes like scraping multiple pages.
* **Maintenance:** Provides system-level visibility for developers working on different modules.

**# App.py :** <https://abseil.io/docs/python/guides/app>

**Purpose:** It serves as the main entry point for the application, integrating all components of the text analysis pipeline. It handles user input, processes queries using the BM25 model, and returns relevant answers with associated sentiment. It also initializes required resources such as the cleaned data, sentiment scores, and BM25 index.

**Key Features:**

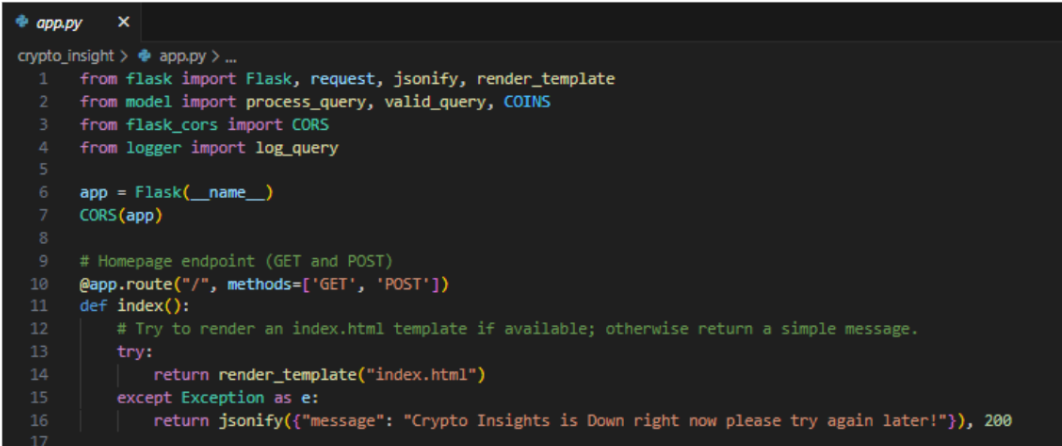
* **User Interaction:** Accepts user queries (via terminal or web interface) and returns ranked answers.
* **BM25 Integration:** Leverages the BM25 model to retrieve the most relevant sentences from the processed dataset.
* **Sentiment Display:** Enhances answers with sentiment scores and categories (positive, negative, neutral).
* **Modular Calls:** Ties together preprocessed text, BM25 search, and sentiment data from other modules.
* **Execution Control:** Acts as the main script run to launch the full pipeline or demo the system.

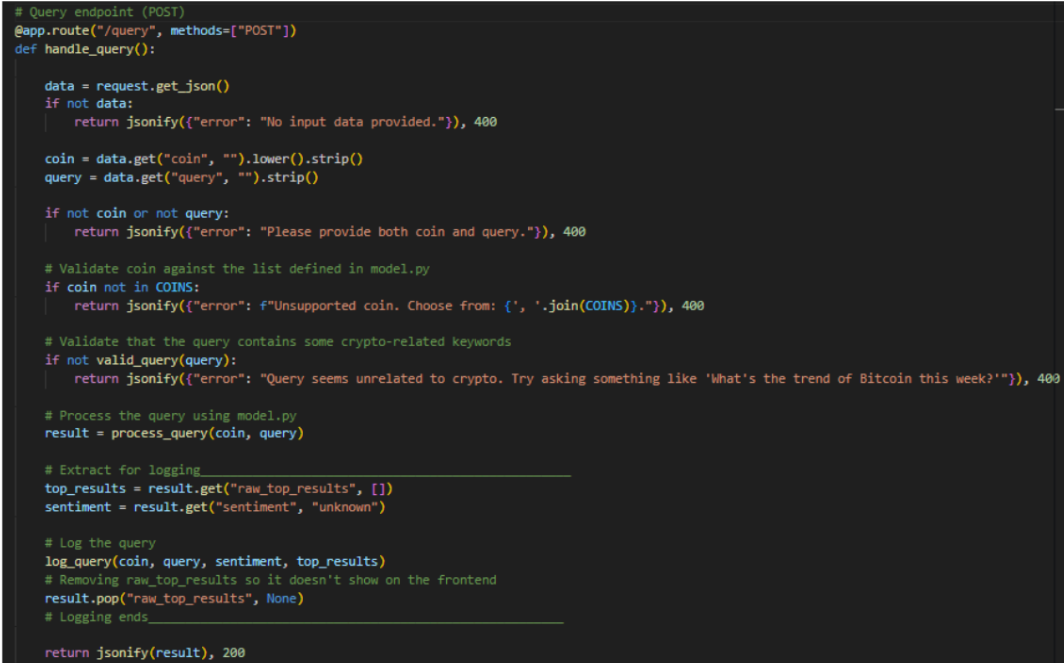
**Usage in code:**

* Executes BM25 search to retrieve and rank the top N relevant sentences.
* Displays matched sentences along with metadata like BM25 score, sentiment polarity, and article source.
* Optionally logs the query and results into query\_logs.json for tracking and debugging.

**Importance:**

* **Integration Point:** Bridges data preprocessing, keyword relevance, and sentiment analysis into one cohesive flow.
* **User Interface Layer:** Acts as the main access point for users to interact with the system.
* **Testing & Deployment:** Ideal for quickly testing the pipeline with different queries or deploying a minimal prototype.





**app.py (continued)**

**Libraries Used:**

* [**Flask**](https://flask.palletsprojects.com/)**:**  
  A micro web framework for Python. Enables routing, rendering HTML templates, handling HTTP requests (GET, POST), and serving JSON responses.  
  Used here to define two main endpoints:
  + / – serves the frontend index.html
  + /query – accepts JSON input (coin + query) and returns results
* [**Flask-CORS**](https://flask-cors.readthedocs.io/en/latest/)**:**  
  Adds Cross-Origin Resource Sharing (CORS) headers to Flask responses. This allows frontend scripts from different domains (or file origins) to communicate with the Flask backend without security blocks.  
  Used via CORS(app) to support frontend–backend communication.
* [**render\_template**](https://flask.palletsprojects.com/en/2.3.x/api/#flask.render_template)**:**  
  Renders HTML files from the templates/ folder. Used to serve index.html.
* [**request.get\_json()**](https://flask.palletsprojects.com/en/2.3.x/api/#flask.Request.get_json)**:**  
  Parses incoming POST requests with JSON payloads into Python dictionaries. Used to fetch user input from the frontend (coin and query).

**Endpoint Logic Breakdown:**

**/ Route (GET)**

@app.route("/", methods=['GET', 'POST'])

def index():

try:

return render\_template("index.html")

except Exception as e:

return jsonify({"message": "Crypto Insights is Down right now please try again later!"}), 200

* Loads the interface via index.html from the templates/ directory.
* Fallbacks to a JSON message in case of errors (e.g., missing template).

**/query Route (POST)**

Handles the backend logic:

@app.route("/query", methods=["POST"])

def handle\_query():

**Input Handling and Validation:**

data = request.get\_json()

if not data:

return jsonify({"error": "No input data provided."}), 400

* Validates presence of data from request body.

**Coin and Query Validation:**

coin = data.get("coin", "").lower().strip()

query = data.get("query", "").strip()

* Strips leading/trailing spaces
* Normalizes to lowercase
* Validates against supported coins defined in model.py
* Ensures query relevance using keywords (valid\_query(query))

**Query Processing:**

result = process\_query(coin, query)

* Calls core logic from model.py
* Returns insights including sentiment and top-ranked responses

**Logging User Interaction:**

log\_query(coin, query, sentiment, top\_results)

* Delegates logging to logger.py for persistent audit trail.

**Output:**

return jsonify(result), 200

* Returns formatted response to frontend.

**logger.py**

**Purpose:**  
Logs each interaction into a local query\_logs.json file with:

* Timestamp
* Coin name
* User’s query
* Sentiment result
* Top answers with scores

**Libraries Used:**

* [**os**](https://docs.python.org/3/library/os.html)**:** for file path handling
* [**json**](https://docs.python.org/3/library/json.html)**:** to read/write logs
* [**datetime**](https://docs.python.org/3/library/datetime.html)**:** for timestamps

**Key Function:** def log\_query(coin, user\_query, sentiment, top\_results):

* Creates an entry object
* Appends to existing log file or creates a new one
* Maintains logs in list format

**Resilience:**  
Wrapped in a try-except block to handle logging errors without affecting backend flow.

# **Frontend API Interaction – script.js**

The file `script.js` handles form submissions and frontend-to-backend communication. It gathers input values from HTML elements, packages them into a JSON payload, and sends them via the Fetch API to the Flask server.  
Error responses are caught and displayed gracefully. The JavaScript also formats and renders the returned sentiment and answer list in the frontend UI dynamically.

# **UI Structure and Styling – index.html & style.css**

The user interface is built using HTML5, styled with modern CSS for responsiveness and dark mode compatibility.  
Elements include:  
- Coin selection dropdown (`<select>`)  
- Query input field (`<input>`)  
- Submission button (`<button>`)  
- Output div (`<div id="result">`)

**Light Frontend work:**

**script.js**

**Purpose:**  
Bridges frontend form with backend API.

function sendQuery() {

const coin = document.getElementById('coin').value;

const query = document.getElementById('query').value;

...

fetch('/query', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ coin, query })

})

**Core Logic:**

* Sends a POST request to /query
* Parses and displays response
* Handles errors and displays fallback messages

**Helper:**

const capitalize = str => str.charAt(0).toUpperCase() + str.slice(1);

Used to format the coin name before displaying.

**Link to Fetch API Docs:**  
[Fetch API – MDN](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

**index.html and style.css**

**index.html:**  
Standard HTML form including:

* Dropdown for coin selection
* Input field for query
* Submit button to trigger sendQuery()
* Result placeholder

**style.css:**  
Custom styling:

* Dark theme with soft shadows
* Responsive form fields
* Visual feedback on hover

**Links:**

* [HTML Select Element – MDN](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select)
* [CSS Styling Basics – MDN](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/Styling_text)

**Quick Links for reference:**

| **Component** | **Recommended reference Link** |
| --- | --- |
| Flask | <https://flask.palletsprojects.com/> |
| Flask-CORS | <https://flask-cors.readthedocs.io/en/latest/> |
| render\_template | <https://flask.palletsprojects.com/en/2.3.x/api/#flask.render_template> |
| request.get\_json() | <https://flask.palletsprojects.com/en/2.3.x/api/#flask.Request.get_json> |
| JSON module | <https://docs.python.org/3/library/json.html> |
| os module | <https://docs.python.org/3/library/os.html> |
| datetime | <https://docs.python.org/3/library/datetime.html> |
| Fetch API | <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API> |
| HTML form elements | <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input> |
| CSS styling | <https://developer.mozilla.org/en-US/docs/Learn/CSS> |