

A Project Report on
Historical Price Insights

Submitted by:

Anjali Sharma (20EBKCS011)

Jatin (20EBKCS052)

in partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

at



B K BIRLA INSTITUTE OF ENGINEERING & TECHNOLOGY

Pilani, Rajasthan (INDIA) - 333031

(Affiliated to Bikaner Technical University, Bikaner, Rajasthan)

June 2024

DECLARATION

We hereby declare that the project entitled “**Historical Price Insights**” submitted for the B. Tech. (CSE) degree is our original work and the project has not formed the basis for the award of any other degree, or any other similar title.

Signature of the Student

Signature of the Student

Place:

Date:

CERTIFICATE

This is to certify that the project titled “**Historical Price Insights**” is the bonafide work carried out by **Anjali Sharma** (20EBKCS011) & **Jatin** (20EBKCS052), students of B Tech (CSE) of B K Birla Institute of Engineering & Technology, Pilani affiliated to Bikaner Technical University, Bikaner, Rajasthan(India) during the academic year 2023-24, in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** (Computer Science and Engineering) and that the project has not formed the basis for the award previously of any other degree, or any other similar title.

Signature of the Guide

Place:

Date:

ACKNOWLEDGMENT

We would like to express our sincere gratitude to the people who have supported and guided us throughout this project.

First and foremost, we are grateful to Dr Nimish Kumar, Head of the Department of Computer Science Engineering, for providing the necessary resources and infrastructure for this project. His vision and leadership in creating a stimulating academic environment have been instrumental in my learning journey.

We are also deeply indebted to our project guide Dr Sonam Mittal, for their invaluable guidance and support throughout this project. Her expertise, patience, and willingness to share their knowledge have been crucial in shaping this project. Her insightful suggestions and constructive feedback helped us to overcome challenges and refine our approach.

We would like to thank the Director, Principal, faculty members, supporting staff, and all those who have directly or indirectly contributed to their unwavering encouragement and support throughout this endeavor. This project would not have been possible without the contributions of these individuals.

We are truly grateful for their support.

Anjali Sharma (20EBKCS011)

Jatin (20EBKCS052)

ABSTRACT

This project aims to develop a comprehensive web scraping and data management system tailored for gathering, storing, and presenting historical commodity data. Leveraging a combination of Python, Selenium, BeautifulSoup, MongoDB, Flask, and Bootstrap, the project provides a robust platform for retrieving, processing, and visualizing commodity prices and trends over the past 30 years.

The project begins by implementing advanced web scraping techniques to extract data from online commodity platforms. Using Selenium for browser automation and BeautifulSoup for HTML parsing, the system retrieves detailed commodity information, ensuring accuracy and completeness. This data is then cleaned and stored in MongoDB, a NoSQL database that offers flexible schema design and efficient data retrieval capabilities.

A Flask-based web application forms the core of the user interface, delivering a seamless and interactive experience for users. Integrated with Bootstrap for responsive design, the web application allows users to explore commodity data through intuitive navigation and dynamic content rendering. Selenium is further employed to automate and enhance user interactions, providing real-time data updates and a realistic browsing experience.

The project also emphasizes rigorous testing methodologies, including unit, integration, and end-to-end testing, to ensure the reliability and functionality of the system. Various deployment strategies, such as cloud deployment and containerization, are considered to ensure scalability, performance, and ease of maintenance.

Ultimately, this project provides a valuable tool for analysts, traders, and researchers to access and analyze historical commodity data. It lays a solid foundation for advanced data analysis and visualization, enabling users to make informed decisions based on comprehensive and up-to-date commodity information.

TABLE OF CONTENTS

CONTENT	PAGE
1. Introduction	1
1.1. Project Overview	1
1.2. Motivation and Problem Statement	1
1.3. Objective and Scope	2
1.4. Technologies Used	2
1.5. Significance of Project	3
2. Literature Review	4
2.1. Related Work in Web Scraping	4
2.2. Existing Commodity Price Data Sources	6
2.3. Comparison of Existing Approaches	7
2.4. Justification of Chosen technologies	8
3. Methodology	9
Structure of Project	9
3.1. Web Scraping using Python	10
3.1.1. Target Website Selection	10
3.1.2. Web Page Structure Analysis	11
3.1.3. Extracting Price Data	12
3.1.4. Error handling and Data Validation	13
3.2. Data Storage with MongoDB	15
3.2.1. MongoDB Database Design	15
3.2.2. Connecting to MongoDB from Python	16
3.2.3. Data Transformation and Uploading	17
3.3. Web Application Development using Flask	19
3.3.1. Flask Project Structure and Environment Setup	19
3.3.2. Routing and handling User Requests	21
3.3.3. Data Retrieval from MongoDB	22
4. Results and Discussions	25
4.1. Data Scraping Performance Evaluation	25
4.1.1. Accuracy and completeness of Scraped Data	25
4.1.2. Efficiency and Scalability	26
4.2. Analysis of Extracted Commodities Price History	27
4.2.1. Descriptive Statistics	28
4.2.2. Visualization of Trends	28
4.2.3. Insight of Observation	29
4.3. Website Functionalities demonstration	30
4.3.1. User Interface Screenshots	30
4.3.2. Explanation of Key Features and User Experience	33
5. Conclusions and Future Work	34
References	38

TABLE OF FIGURES

Figure Number	Figure	Page Number
Figure 3.1	Website to be scraped	11
Figure 3.2	Scraper Code	13
Figure 3.3	Scraper with error handling	14
Figure 3.4	MongoDB collections	16
Figure 3.5	App structure	20
Figure 3.6	Flask Main file	21
Figure 3.7	Data Retrieval from MongoDB	23
Figure 4.1	Efficient data uploading	26
Figure 4.2	Remove null values	27
Figure 4.3	Homepage	30
Figure 4.4	Commodities homepage	30
Figure 4.5	All commodities in a table	31
Figure 4.6	Specific commodity view	31
Figure 4.7	Dynamic Interaction with webpage	32
Figure 4.8	Interactable navbar	33
Figure 5.1	Pipeline and structure of app	34
Figure 5.2	NoSQL database	35

1| INTRODUCTION

The project at hand aims to develop a robust web scraping and data management system tailored for gathering, storing, and presenting historical commodity data sourced from online platforms. Specifically, the system focuses on retrieving and processing data spanning the last 30 years for commodities such as gold. The core functionality involves web scraping using Selenium and BeautifulSoup, data storage in MongoDB, and a user-friendly web interface built with Flask and Bootstrap.

1.1 Project Overview:

- Briefly introduce the project's objective: developing a web scraper for commodity price history from a target website (IndexMundi).
- Highlight the significance of commodity price data and its applications in various sectors (e.g., finance, economics, supply chain management).

1.2 Motivation and Problem Statement:

In the dynamic and volatile world of commodity trading, having access to accurate, comprehensive, and timely data is crucial for making informed decisions. Commodity prices fluctuate due to various factors such as market demand, geopolitical events, currency fluctuations, and natural disasters. Traders, analysts, and researchers rely heavily on historical data to identify trends, forecast future prices, and develop trading strategies.

However, obtaining and managing such data poses significant challenges:

1. **Data Accessibility:** Reliable and detailed historical commodity data is often dispersed across multiple platforms, websites, and formats, making it difficult for users to access and consolidate the information they need efficiently.
2. **Data Integrity:** Ensuring the accuracy and completeness of collected data is a major concern. Inconsistencies, missing values, and incorrect entries can undermine the reliability of analysis and decision-making processes.
3. **Real-time Updates:** The commodity market is highly dynamic, with prices changing rapidly. Traditional methods of data collection and analysis often fail to provide real-time updates, resulting in outdated information that can mislead traders and analysts.
4. **User Experience:** Many existing platforms for commodity data lack user-friendly interfaces and intuitive navigation, making it challenging for users to effectively explore and analyze the data.
5. **Scalability and Maintenance:** As the volume of data grows, managing and scaling the infrastructure to handle large datasets becomes increasingly complex. Ensuring that the system remains performant and easy to maintain is crucial for long-term usability.

1.3 Objective and Scope:

The primary objectives of the project include:

- **Data Collection:** Gather historical commodity data from online sources, ensuring accuracy and completeness.
- **Data Management:** Establish a structured database in MongoDB to store the collected data efficiently.
- **Web Application Development:** Create an intuitive web application to present the commodity data in an organized and visually appealing manner.
- **Regular Updates:** Implement mechanisms to ensure that the data is regularly updated to reflect the latest information available.
- **User Experience:** Prioritize a seamless user experience by designing a responsive and user-friendly interface for data exploration.

Scope of the project:

- Specify the types of commodities to be scraped (e.g., metals, energy, agriculture).
- Clarify the time period for which historical data will be collected.
- Mention any limitations or functionalities not included in this project (e.g., data analysis tools, user authentication).

1.4 Technologies Used:

The project leverages the following technologies to achieve its objectives:

- **Python:** Python is known for its versatility in scripting, web scraping (using libraries like Selenium and BeautifulSoup), and data manipulation. It's the foundation for building the data collection and processing functionalities.
- **Selenium:** Handles browser interaction and retrieves data from dynamic web pages that might not be readily accessible to traditional scraping methods.
- **BeautifulSoup:** Parses the extracted HTML content, allowing you to pinpoint and extract the relevant data efficiently.
- **MongoDB:** Advantages of MongoDB in storing large datasets:
 - **Flexibility:** Supports various data structures, making it suitable for storing diverse commodity data.
 - **Scalability:** Can handle growing data volumes efficiently, ensuring smooth operation as the data collection expands.

- **Flask:** It is a lightweight web framework for building the user interface:
 - **Rapid Development:** Flask allows for quick development of web applications, making it efficient for building the user interface.
 - **Easy Integration:** Integrates well with Python code, simplifying the process of connecting the backend data manipulation to the user interface.
- **Bootstrap:** It enhances the user experience:
 - **Responsiveness:** Bootstrap makes the web application responsive, meaning it adjusts to different screen sizes and devices, providing a seamless experience for users.
 - **Visually Appealing Design:** Bootstrap offers pre-built styles and components, allowing you to create a user-friendly and visually appealing interface for data exploration.

1.5 Significance of Project:

The key advantages of project:

- **Efficiency:** Users can access a vast amount of historical commodity data in one place, saving them time and resources.
- **Reliability:** Automated data collection reduces errors and enhances data accuracy, leading to more reliable analysis.
- **Flexibility:** The web application allows users to explore data for different commodities and time periods, providing them with greater flexibility.
- **Informed Decision-Making:** By enabling users to visualize trends and identify patterns, the project empowers users with insights for making informed data-driven decisions.

2] Literature Review

This chapter delves into the existing body of research and technological landscape relevant to your project of collecting commodity price data. It aims to establish a solid foundation by exploring various data acquisition methods, identifying challenges and limitations, and ultimately justifying the chosen methodology and tools for your project.

2.1 Related Work in Web Scraping: Techniques and Considerations

Web scraping, the process of extracting data from websites, has a rich body of related work spanning multiple fields including computer science, data science, and information retrieval. Researchers and practitioners have developed various techniques and tools to effectively extract, process, and utilize web data.

- **Web Scraping Techniques:** This section dissects the various methods employed to acquire data from websites, highlighting their relative merits in the context of your commodity price collection project:
 - **Browser Automation Tools:** Discuss the utilization of tools like Selenium to emulate web browsers and interact with websites dynamically. This might be necessary for scraping content that is user-driven (e.g., interactive charts) or requires login credentials (e.g., member-only data). Emphasize the importance of employing Selenium responsibly by mimicking human behavior and adhering to website scraping limitations. Highlight the benefits of responsible scraping, such as avoiding server overload and respecting website terms of service.
 - **API Integration:** Explore the potential use of Application Programming Interfaces (APIs) offered by some websites to access data programmatically. This is generally the preferred approach if available, as it offers structured data formats and clear documentation, simplifying data collection and processing. Additionally, API usage adheres to ethical scraping practices by utilizing a sanctioned access method.
 - **Static Content Scraping:** Explain how libraries like BeautifulSoup can be leveraged to parse HTML code and extract data from static web

pages. This approach suits websites with relatively stable layouts and content structures. Consider mentioning the strengths of BeautifulSoup, such as its ease of use and versatility in handling various HTML parsing needs, making it a valuable tool for acquiring basic commodity price information.

- **Common Web Scraping Tools and Libraries:** Briefly introduce popular libraries and tools used for web scraping, including advantages and disadvantages of potential alternatives to your chosen tools. For instance, compare BeautifulSoup with Scrapy, a more robust framework for large-scale scraping projects. Highlight the trade-off between ease of use (BeautifulSoup) and the power and efficiency of Scrapy for complex scraping tasks.
- **Challenges and Mitigation Strategies:** Discuss the various challenges encountered during web scraping, along with potential mitigation strategies to ensure successful data collection for your commodity price project:
 - **Website Structure Dynamics:** Websites often undergo layout and code changes, which can disrupt scraping scripts reliant on specific HTML elements. Emphasize the importance of writing adaptable scraping code that can handle minor structural changes. This might involve using CSS selectors or regular expressions for flexible pattern matching within your scraping logic.
 - **Anti-Scraping Measures:** Websites may implement measures to detect and prevent automated scraping, requiring adaptation of scraping techniques. Discuss potential anti-scraping measures like CAPTCHAs or IP blocking, and propose how your approach might address them (e.g., using headless browsers to avoid detection or rotating IP addresses to prevent blocking). Reiterate the importance of respecting robots.txt files, which specify scraping limitations set by website owners.
 - **Ethical Considerations:** Emphasize the importance of adhering to ethical scraping practices. You can elaborate on specific ethical guidelines, such as throttling scraping requests to avoid overwhelming servers, complying with website terms of service, and considering the legal implications of scraping data from certain websites. Providing

citations to relevant resources on ethical scraping best practices can strengthen this section.

- **References:** Provide a comprehensive list of references to relevant research papers or articles on web scraping techniques, ethical considerations, and any specific libraries or tools you mentioned in this section.

2.2 Existing Commodity Price Data Sources: Unveiling the Landscape

Commodity price data is crucial for various stakeholders including traders, analysts, and policymakers. There are several established sources from which commodity price data can be obtained:

- **Government and International Agencies:** Organizations such as the United States Department of Agriculture (USDA), Food and Agriculture Organization (FAO), and World Bank provide extensive datasets on commodity prices. These sources are highly reliable and are often used for policy-making and economic analysis.
- **Marketplaces and Exchanges:** Commodity exchanges like the Chicago Mercantile Exchange (CME), New York Mercantile Exchange (NYMEX), and London Metal Exchange (LME) offer real-time and historical price data. These sources are essential for traders and investors.
- **Financial News and Data Providers:** Platforms like Bloomberg, Reuters, and Yahoo Finance offer commodity price data along with news and analysis. These sources are valuable for staying updated with market trends and gaining insights.
- **E-commerce Websites:** Websites such as Amazon and Alibaba also provide price data, though more focused on retail and wholesale products rather than raw commodities.
- **Specialized Commodity Data Providers:** Companies like MetalMiner and Trading Economics specialize in providing detailed commodity price data and analysis.

2.3 Comparison of Existing Approaches and Limitations

Different approaches to web scraping and data extraction come with their own sets of advantages and limitations.

Here is a comparison of some popular methods:

- **Regular Expressions:**
 - *Advantages:* Lightweight, fast, and suitable for simple HTML structures.
 - *Limitations:* Not scalable for complex or nested HTML, difficult to maintain, and error-prone with changes in the web page structure.
- **BeautifulSoup:**
 - *Advantages:* Easy to use, well-documented, handles complex HTML structures, integrates well with other Python libraries.
 - *Limitations:* Slower for large datasets compared to more specialized frameworks, primarily designed for parsing rather than web crawling.
- **Scrapy:**
 - *Advantages:* Comprehensive framework for large-scale web scraping, supports request handling, link following, and data pipelines, highly customizable.
 - *Limitations:* Steeper learning curve, may be overkill for small projects, requires more setup and configuration.
- **Selenium:**
 - *Advantages:* Capable of interacting with JavaScript-heavy websites, simulates real user interactions, supports multiple browsers.
 - *Limitations:* Resource-intensive, slower execution, requires a browser driver, more complex to set up and use compared to headless scraping tools.
- **API-based Data Extraction:**
 - *Advantages:* Reliable and structured data, respects website's terms of service, often includes comprehensive documentation and support.
 - *Limitations:* Limited by API rate limits, requires API access permissions, not all data may be available through APIs.

2.4 Justification of Chosen Technologies

When selecting technologies for web scraping and data processing, several factors including project requirements, ease of use, community support, and scalability must be considered. Here's why the chosen technologies are suitable:

- **Python:**

Python is a versatile and powerful programming language with a rich ecosystem of libraries for web scraping, data processing, and web development. Its readability and ease of learning make it accessible for both beginners and experienced developers.

- **BeautifulSoup:**

BeautifulSoup is chosen for its simplicity and efficiency in parsing HTML and XML documents. It is ideal for projects where data extraction from well-structured HTML is required. Its integration with Python makes it easy to use and highly effective for small to medium-sized scraping tasks.

- **Selenium:**

Selenium is selected for its capability to handle JavaScript-heavy websites. It allows for the simulation of user interactions in a real browser, making it possible to scrape data from dynamic web pages. This is particularly useful for websites that rely on client-side rendering.

- **Flask:**

Flask is a lightweight web framework for Python that enables quick and easy development of web applications. It is well-suited for building APIs and simple web interfaces, making it a good choice for developing the backend of a web scraping application.

- **Bootstrap:**

Bootstrap is a popular front-end framework that simplifies the development of responsive and aesthetically pleasing web interfaces. Its extensive library of components and pre-designed templates allows for rapid development and consistent design across the web application.

- **MongoDB:**

MongoDB is a NoSQL database known for its flexibility and scalability. Its document-oriented structure makes it suitable for storing unstructured or semi-structured data commonly obtained from web scraping. It also supports horizontal scaling and handles large volumes of data efficiently.

3] Methodology

This chapter delves into the specific methods and technologies employed to collect, store, and visualize commodity price data for our project. Here, we'll explore the utilization of Python for web scraping, MongoDB for data storage, and Flask with Bootstrap for developing a user-friendly web application.

Structure of Project:

```
| - College Project/  
  | - analysis/  
    | | - clean_homepage.py  
    |  
  | - database/  
    | | - clean_homepage.py  
    | | - createDB_commodity.py  
    | | - createDB_home.py  
    |  
  | - files/  
    | | - all_commodities.csv  
    | | - clean_all_commodities.csv  
    |  
  | - scrapers/  
    | | - commodity_scraper.py  
    | | - currency.py  
    | | - homepage_scrapeer.py  
    | | - max_years.py  
    |  
  | - static/  
    | | - assets/  
    | | | - img/  
    | | | | - bg.png  
    | | | - visuals/
```


- | | | |- png files
- | | |- favicon.ico
- | |- css/
- | | |- styles.css
- | |- js/
- | | |- scripts.js
- |
- |- templates/
- | |- about.html
- | |- all_commodities.html
- | |- commodity.html
- | |- contactt.html
- | |- footer.html
- | |- header.html
- | |- index.html
- |
- |- main.py
- |- README.md
- |- requirements.txt
- |- structure.txt

3.1 Web Scraping with Python

Web scraping is the process of automatically extracting data from websites using software tools. In this project, Python is chosen as the programming language for building the web scraper due to its versatility, ease of use, and the availability of robust libraries for web scraping and data manipulation.

3.1.1 Target Website Selection (IndexMundi)

The first step in any web scraping project is to identify the target website from which data needs to be extracted.

In this case, IndexMundi (<https://www.indexmundi.com/>) has been selected as the target website. IndexMundi is a data portal that provides comprehensive information

and statistics on various topics, including commodity prices, demographics, economy, and more.

The decision to choose IndexMundi as the target website is based on several factors:

1. **Data Relevance:** IndexMundi provides up-to-date and comprehensive data on commodity prices, which is the primary focus of this project.
2. **Data Quality:** The data available on IndexMundi is sourced from reputable international organizations and government agencies, ensuring a high level of data quality and reliability.
3. **Website Structure:** The IndexMundi website has a well-organized structure, making it easier to navigate and locate the desired data.
4. **Legal Considerations:** IndexMundi's terms of service generally allow web scraping for non-commercial purposes, provided that certain guidelines are followed, such as not overloading the website with excessive requests.

By selecting IndexMundi as the target website, this project aims to leverage the wealth of commodity price data available, while adhering to ethical and legal practices in web scraping.

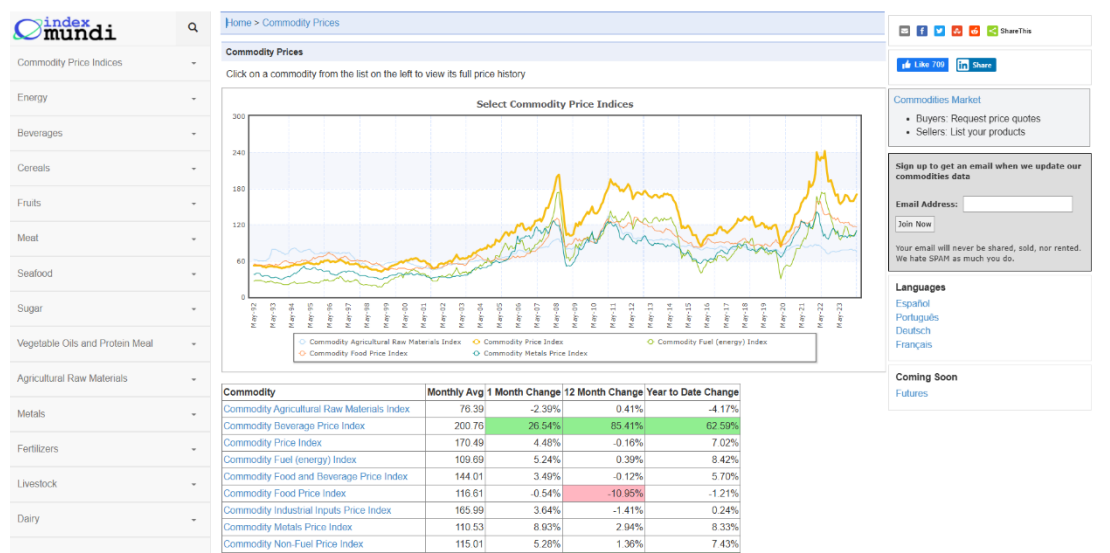


Figure 3.1: Website to be scraped

3.1.2 Web Page Structure Analysis

Before diving into the web scraping code, it is crucial to analyze the structure of the web pages from which data needs to be extracted. This analysis involves understanding the HTML markup, identifying the relevant HTML tags and elements that contain the desired data, and determining the patterns or rules that govern the data presentation.

To analyze the web page structure, the following steps can be taken:

1. **Inspect the Web Page:** Open the target web page in a web browser and use the built-in developer tools (e.g., Chrome DevTools, Firefox Developer Tools) to inspect the HTML structure of the page.

2. **Locate the Data Elements:** Within the developer tools, navigate through the HTML elements and identify the specific tags or elements that contain the desired data, such as commodity names, prices, and other relevant information.
3. **Understand the Data Patterns:** Observe the patterns or rules that govern the placement and organization of the data elements within the HTML structure. This may involve analyzing the class names, id attributes, or other identifying characteristics of the relevant elements.
4. **Test Across Multiple Pages:** If the data is spread across multiple web pages, analyze the structure and patterns across several pages to ensure consistency and identify any variations that need to be handled in the scraping code.

Thorough web page structure analysis is crucial for writing efficient and targeted web scraping code. It helps identify the specific HTML elements and patterns to target, reducing the likelihood of errors and ensuring accurate data extraction.

3.1.3 Extracting Price Data Using BeautifulSoup

Once the target website and web page structure have been analyzed, the next step is to write the code to extract the desired data using Python and the BeautifulSoup library.

BeautifulSoup is a powerful Python library that provides a convenient way to parse HTML and XML documents. It creates a parse tree for the HTML or XML input, allowing developers to navigate and search the tree structure using various methods and attributes.

The process of extracting price data using BeautifulSoup typically involves the following steps:

1. **Import Required Libraries:** Import the necessary Python libraries, including BeautifulSoup and any additional libraries required for making HTTP requests (e.g., `requests` library).
2. **Send HTTP Request:** Use the `requests` library or another HTTP client library to send a GET request to the target URL and retrieve the HTML content of the web page.
3. **Parse the HTML:** Create a BeautifulSoup object by passing the retrieved HTML content to the BeautifulSoup constructor. This will parse the HTML and create a parse tree that can be navigated and searched.
4. **Locate Data Elements:** Use BeautifulSoup's selection methods, such as `find()`, `find_all()`, or CSS selectors, to locate the specific HTML elements that contain the desired data (e.g., commodity names, prices).
5. **Extract Data:** Once the relevant elements are located, extract the desired data from the element's text or attribute values using BeautifulSoup's methods and attributes.
6. **Store or Process Data:** Depending on the project requirements, the extracted data can be stored in a data structure (e.g., lists, dictionaries) for further processing or directly written to a file or database.

```

1 import csv
2 import json
3 from bs4 import BeautifulSoup
4 from selenium import webdriver
5 from selenium.webdriver.chrome.options import Options
6
7 def home_scraper():
8     chrome_options = Options()
9     driver = webdriver.Chrome(options=chrome_options)
10
11     records = []
12     url = "https://www.indexmundi.com/commodities/"
13
14     driver.get(url)
15     soup = BeautifulSoup(driver.page_source, 'html.parser')
16     parent_commodities = soup.find('table', 'tblData')
17     all_commodities = parent_commodities.find_all('tr')
18
19     for item in all_commodities:
20         try:
21             temp = []
22             a_tag = item.find('td').a
23             link = a_tag.get('href')
24
25             all_values = item.find_all('td')
26             for i in all_values:
27                 temp.append(i.text)
28
29             link = url + link
30             temp.append(link)
31             records.append(temp)
32
33         except Exception as e:
34             print(f"Error processing item: {e}")
35             continue
36
37     driver.quit()
38
39     # Write records to CSV
40     with open('files/all_commodities.csv', 'w', newline='', encoding='utf-8') as csv_file:
41         writer = csv.writer(csv_file)
42         writer.writerow(['Commodity Name', 'Monthly Avg', '1 Month Change', '12 Month Change', 'Year to Date Change', 'URL'])
43         writer.writerows(records)
44
45

```

Figure 3.2: Scraper code

This example demonstrates how BeautifulSoup can be used to locate and extract data from specific HTML elements based on their tags, classes, or other attributes. However, the actual implementation will depend on the structure and patterns of the IndexMundi web pages and the desired data to be extracted.

3.1.4 Error Handling and Data Validation

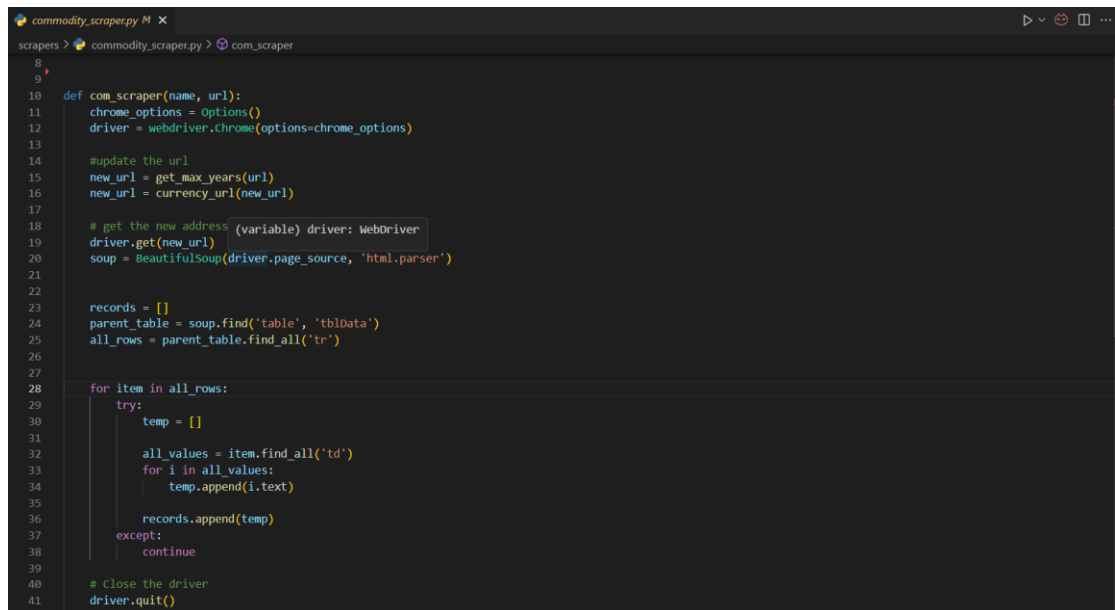
Web scraping can be prone to various errors and exceptions due to factors such as network issues, website changes, or unexpected data formats. To ensure a robust and reliable web scraping process, it is crucial to implement proper error handling and data validation mechanisms.

Error Handling

Error handling involves anticipating and gracefully handling potential errors or exceptions that may occur during the web scraping process. This can be achieved through the following techniques:

1. **Exception Handling:** Use Python's built-in exception handling mechanisms (try-except blocks) to catch and handle specific exceptions that may be raised during the web scraping process. For example, handle exceptions related to network connectivity issues, timeouts, or HTTP errors.
2. **Retry Mechanisms:** Implement retry mechanisms to automatically attempt to re-establish connections or re-send requests in case of temporary failures or network issues.
3. **Logging:** Incorporate logging mechanisms to record errors, warnings, and other relevant information during the web scraping process. This can help in identifying and troubleshooting issues more effectively.

4. **Graceful Termination:** Ensure that the web scraping process terminates gracefully in case of unrecoverable errors, releasing any acquired resources (e.g., network connections, file handles) and providing informative error messages or notifications.



```
8
9
10 def com_scraper(name, url):
11     chrome_options = Options()
12     driver = webdriver.Chrome(options=chrome_options)
13
14     #update the url
15     new_url = get_max_years(url)
16     new_url = currency_url(new_url)
17
18     # get the new address (variable) driver: WebDriver
19     driver.get(new_url)
20     soup = BeautifulSoup(driver.page_source, 'html.parser')
21
22
23     records = []
24     parent_table = soup.find('table', 'tblData')
25     all_rows = parent_table.find_all('tr')
26
27
28     for item in all_rows:
29         try:
30             temp = []
31
32             all_values = item.find_all('td')
33             for i in all_values:
34                 temp.append(i.text)
35
36             records.append(temp)
37         except:
38             continue
39
40     # Close the driver
41     driver.quit()
```

Figure 3.3: Scraper with error handling

Data Validation

In addition to error handling, it is important to validate the extracted data to ensure its integrity and consistency. Data validation can involve the following steps:

1. **Data Type Validation:** Verify that the extracted data is of the expected data type (e.g., string, float, integer) and take appropriate actions if the data is of an unexpected type.
2. **Value Range Validation:** Check if the extracted data falls within expected value ranges or adheres to predefined constraints (e.g., prices should be positive numbers).
3. **Format Validation:** Ensure that the extracted data conforms to specific formats or patterns (e.g., date formats, currency formats).
4. **Data Completeness Validation:** Verify that all the required fields or data elements have been successfully extracted and that there are no missing or incomplete data points.
5. **Data Consistency Validation:** Check for consistency across related data points or fields (e.g., ensure that prices are consistent across different sections of the website).

3.2 Data Storage with MongoDB

MongoDB is a popular NoSQL database that provides a flexible and scalable solution for storing and managing data. In this project, MongoDB is chosen as the database for storing the extracted commodity price data due to its ability to handle semi-structured and unstructured data, its horizontal scalability, and its rich querying capabilities.

3.2.1 MongoDB Database Design

Before storing data in MongoDB, it is essential to design an appropriate data model that aligns with the application's requirements and ensures efficient data storage and retrieval. The database design process for MongoDB involves the following steps:

1. **Data Analysis:** Analyze the structure and characteristics of the data to be stored. In this case, the data consists of commodity prices, which may include fields such as commodity name, price, currency, unit, and potentially other related information.
2. **Document Structure Design:** In MongoDB, data is stored in JSON-like documents, which are analogous to rows in a relational database. Determine the structure of the documents that will represent the commodity price data. This may involve nesting related information within subdocuments or embedding data for better query performance.
3. **Collection Design:** MongoDB organizes documents into collections, which are similar to tables in a relational database. Decide on the appropriate collection(s) to store the commodity price data. Factors to consider include the anticipated data volume, access patterns, and potential for data partitioning or sharding.
4. **Schema Design (Optional):** MongoDB is a schema-less database, meaning that documents within a collection can have different structures. However, for data consistency and validation purposes, it is often beneficial to define a schema using tools like MongoDB's JSON Schema or third-party libraries like Mongoose for Node.js or PyMongo for Python.
5. **Indexing Strategy:** Determine the appropriate indexes to create on the collections to optimize query performance. Indexes in MongoDB can be created on single fields or compound fields, depending on the anticipated queries and access patterns.
6. **Data Relationships:** Analyze any potential relationships between different data entities (e.g., commodity prices and commodity categories) and decide on the appropriate data modeling approach, such as embedding or referencing.
7. **Security and Access Control:** Define security measures, such as authentication mechanisms, access control rules, and data encryption strategies, to ensure the protection and integrity of the stored data.

Proper database design is crucial for efficient data storage, retrieval, and management in MongoDB. It also helps in ensuring data consistency, maintainability, and scalability of the application.

The screenshot shows the MongoDB Atlas interface for a project named 'college_project'. The 'Data Services' tab is active, displaying the 'commodityDB' database. The database has 13 collections and 1 database. The interface includes a sidebar with navigation options like Overview, Deployment, Database, Data Lake, Services, and Security. The main area shows a table of collections with their respective statistics.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
Aluminum	352	34.74KB	102B	32KB	1	28KB	28KB
Coal, Australian thermal coal	350	41.26KB	121B	32KB	1	28KB	28KB
Coffee, Other Mild Arabicas	350	40.68KB	120B	32KB	1	28KB	28KB
Cotton	350	32.45KB	95B	48KB	1	44KB	44KB
Crude Oil (petroleum), Price index	350	43.27KB	127B	52KB	1	44KB	44KB
Diesel	213	19.82KB	96B	24KB	1	24KB	24KB
Gold	1400	132.08KB	97B	108KB	1	76KB	76KB
Indonesian Liquefied Natural Gas	350	41.59KB	122B	32KB	1	28KB	28KB
Jet Fuel	352	33.36KB	98B	32KB	1	28KB	28KB
Lead	350	33.16KB	98B	52KB	1	48KB	48KB

Figure 3.4: MongoDB collections

3.2.2 Connecting to MongoDB from Python

To interact with MongoDB from a Python application, the PyMongo library is commonly used. PyMongo is the official Python driver for MongoDB, providing a convenient and idiomatic way to interact with MongoDB databases.

The process of connecting to MongoDB from Python using PyMongo typically involves the following steps:

1. **Install PyMongo:** If not already installed, install the PyMongo library using a package manager like `pip`:

```
Pip install pymongo
```

2. **Import Required Modules:** In the Python script or application, import the necessary modules from PyMongo:

```
from pymongo import MongoClient
```

3. **Create a MongoDB Connection:** Create a `MongoClient` instance, specifying the connection details such as the MongoDB server address and port:

```
client = MongoClient('mongodb://localhost:27017/')

```

```
# Connect to a remote MongoDB instance client =
MongoClient('mongodb://username:password@host:port/')

```

4. **Access the Database:** Once the connection is established, access the desired database using the client instance:

```
# Access or create a database
db = client['database_name']

```

5. **Interact with Collections:** Within the database, interact with collections (analogous to tables in a relational database) to perform various operations, such as inserting, querying, updating, or deleting data:

```
# Access or create a collection
collection = db['collection_name']

# Insert a document
document = {'name': 'Commodity A', 'price': 100.0, 'currency': 'USD'}
result = collection.insert_one(document)

# Query documents
query = {'price': {'$gt': 50.0}}
results = collection.find(query)

# Update documents
update_filter = {'name': 'Commodity A'}
update_values = {'$set': {'price': 110.0}}
result = collection.update_many(update_filter, update_values)

# Delete documents
delete_filter = {'price': {'$lt': 20.0}}
result = collection.delete_many(delete_filter)
```

6. **Handle Errors and Exceptions:** Implement error handling and exception management mechanisms to gracefully handle any issues that may arise during the MongoDB interaction, such as connection failures or query errors.
7. **Close the Connection:** Once the required operations are completed, close the MongoDB connection to release resources:

```
Client.close()
```

By following these steps, Python applications can seamlessly connect to MongoDB using PyMongo and perform various data manipulation operations, such as inserting, querying, updating, and deleting data in MongoDB collections.

3.2.3 Data Transformation and Uploading to MongoDB

After extracting the commodity price data using web scraping techniques, it is often necessary to transform the data into a format suitable for storage in MongoDB. This process may involve data cleaning, restructuring, or mapping the extracted data to the designed MongoDB document structure.

The data transformation and uploading process can be divided into the following steps:

1. **Data Preparation:** Depending on the initial format of the extracted data, perform any necessary data cleaning or preprocessing steps. This may include

removing unnecessary fields, handling missing values, converting data types, or applying data normalization techniques.

2. **Data Mapping:** Map the extracted and cleaned data to the designed MongoDB document structure. This step involves creating dictionaries or lists of dictionaries that conform to the MongoDB document structure defined during the database design phase.
3. **Connect to MongoDB:** Establish a connection to the MongoDB instance using PyMongo, as described in the previous section (3.2.2 Connecting to MongoDB from Python).
4. **Insert or Update Data:** Once the data is mapped to the appropriate document structure, use PyMongo's methods to insert or update the data in the corresponding MongoDB collection.
 - **Inserting Data:** If the data represents new records, use the `insert_one()` or `insert_many()` methods to insert individual documents or a list of documents, respectively:

```
result = collection.insert_one(document)
```

```
result = collection.insert_many(documents_list)
```

- **Updating Data:** If the data represents updates to existing records, use the `update_one()` or `update_many()` methods along with appropriate filters and update operators:

```
update_filter = {'commodity_name': 'Wheat'}
```

```
update_values = {'$set': {'price': 120.0}}
```

```
result = collection.update_one(update_filter,  
update_values)
```

5. **Data Validation:** Implement data validation mechanisms to ensure that the transformed data adheres to the defined MongoDB schema (if applicable) and meets any additional data integrity constraints.
6. **Error Handling:** Incorporate error handling and exception management mechanisms to gracefully handle any issues that may arise during the data transformation or uploading process, such as data format errors or MongoDB connection issues.
7. **Close the Connection:** After completing the data upload process, close the MongoDB connection to release resources:

```
Client.close()
```

3.3 Web Application Development with Flask and Bootstrap

In this project, a web application is developed using the Flask web framework and the Bootstrap front-end framework. The purpose of the web application is to provide a user-friendly interface for displaying and interacting with the commodity price data stored in the MongoDB database.

3.3.1 Flask Project Structure and Environment Setup

Flask is a lightweight and flexible Python web framework that follows the Model-View-Controller (MVC) architectural pattern. The project structure for a Flask application typically involves the following components:

1. **Project Directory:** The root directory that contains all the files and directories related to the Flask application.
2. **Application File:** A Python file (e.g., `app.py`) that serves as the entry point for the Flask application and contains the main code for initializing the app, defining routes, and handling requests.
3. **Templates Directory:** A directory (e.g., `templates/`) that contains HTML template files used for rendering the user interface.
4. **Static Directory:** A directory (e.g., `static/`) that stores static files such as CSS stylesheets, JavaScript files, and images.
5. **Configuration Files:** Files (e.g., `config.py`) that store configuration settings for the application, such as database connection details, secret keys, and other environment-specific settings.
6. **Requirements File:** A file (e.g., `requirements.txt`) that lists all the Python dependencies and libraries required by the Flask application, making it easier to set up the project environment.

To set up the project environment, follow these steps:

1. **Create a Virtual Environment:** Use a virtual environment tool like `virtualenv` or `venv` to create an isolated Python environment for the project:

```
python -m venv env
```

2. **Activate the Virtual Environment:** Activate the virtual environment:

On Windows: `env\Scripts\activate`

On Unix or macOS: `source env/bin/activate`

3. **Install Dependencies:** Install the required Python packages and libraries listed in the `requirements.txt` file:

```
pip install -r requirements.txt
```

4. **Set Environment Variables:** Set any necessary environment variables, such as database connection strings or secret keys, either directly in the terminal or in a separate configuration file (e.g., `config.py`).
- 5.

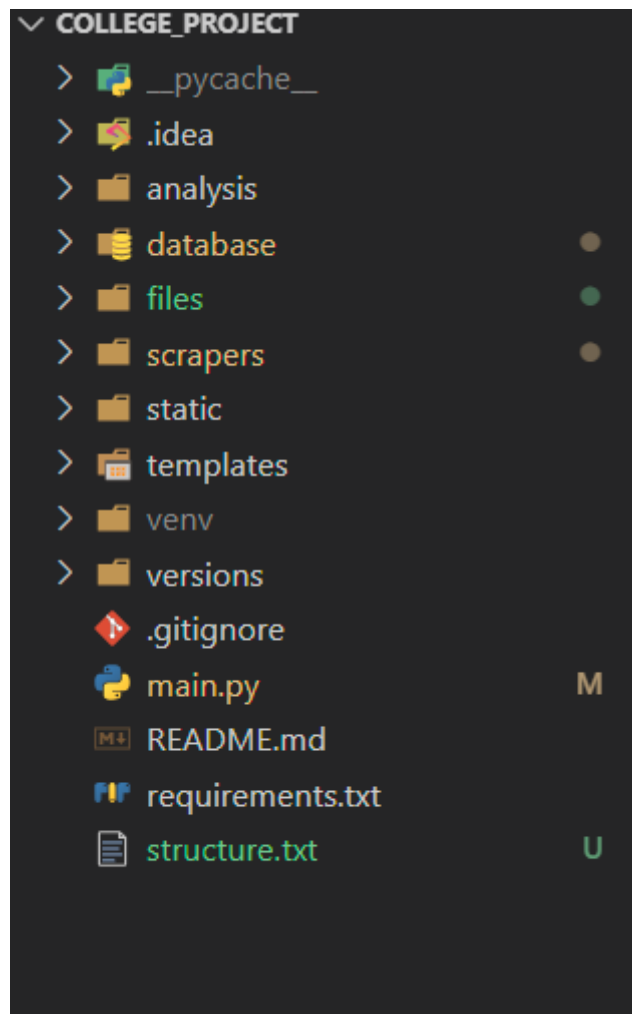


Figure 3.5: App structure

6. **Run the Flask Application:** Start the Flask development server by running the application file (e.g., `app.py`):

```
Flask run
```

3.3.2 Routing and Handling User Requests

Flask uses a routing mechanism to map URL paths to specific Python functions (views) that handle the corresponding HTTP requests. This section covers how to define routes, handle different HTTP methods (GET, POST, etc.), and process user input or form data.

1. **Defining Routes:** In Flask, routes are defined using the `@app.route` decorator. The decorator is applied to a Python function that will handle the requests for the specified URL path.

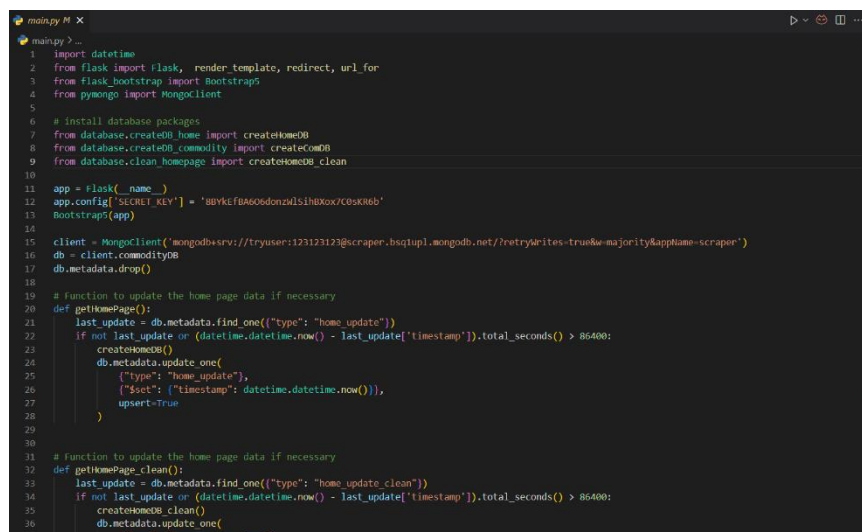
In this example, the `index` function handles requests for the root URL (`/`), and the `commodities` function handles requests for the `/commodities` URL path.

2. **Handling HTTP Methods:** Flask supports different HTTP methods (GET, POST, PUT, DELETE, etc.) for handling various types of requests. The HTTP method can be specified in the `@app.route` decorator using the `methods` parameter.

In this example, the `add_commodity` function handles both GET and POST requests for the `/add_commodity` URL path. If the request method is POST, it processes the form data submitted by the user and saves it to MongoDB. If the request method is GET, it renders the form template.

3. **Processing User Input:** Flask provides access to user input data through the request object. For form data submitted via POST requests, the `request.form` dictionary can be used to access the form fields.

By defining routes, handling HTTP methods, and processing user input, Flask allows you to create dynamic web applications that respond to user requests and interactions.



```
main.py X
main.py > -
1 import datetime
2 from flask import Flask, render_template, redirect, url_for
3 from flask_bootstrap import Bootstrap5
4 from pymongo import MongoClient
5
6 # Install database packages
7 from database.createdb_home import createHomeDB
8 from database.createdb_commodity import createComDB
9 from database.cleanedb_homepage import createHomeDB_clean
10
11 app = Flask(__name__)
12 app.config['SECRET_KEY'] = '8BYkEfBA6O6donzC1sihX0K/CsXK9bb'
13 Bootstrap5(app)
14
15 client = MongoClient('mongodb+srv://tryuser:123123123@scraper.bs4up1.mongodb.net/?retryWrites=true&majority=asptime=scraper')
16 db = client.commodityDB
17 db.metadata.drop()
18
19 # function to update the home page data if necessary
20 def getHomePage():
21     last_update = db.metadata.find_one({"type": "home update"})
22     if not last_update or (datetime.datetime.now() - last_update['timestamp']).total_seconds() > 86400:
23         createHomeDB()
24         db.metadata.update_one(
25             {'type': 'home update'},
26             {'$set': {'timestamp': datetime.datetime.now()}},
27             upsert=True
28         )
29
30 # function to update the home page data if necessary
31 def getHomePage_clean():
32     last_update = db.metadata.find_one({"type": "home update_clean"})
33     if not last_update or (datetime.datetime.now() - last_update['timestamp']).total_seconds() > 86400:
34         createHomeDB_clean()
35         db.metadata.update_one(
36             {'type': 'home update_clean'}
```

Figure 3.6: Flask main file

3.3.3 Data Retrieval from MongoDB for Display

To display the commodity price data stored in MongoDB, the Flask application needs to retrieve the data from the database and pass it to the corresponding templates for rendering.

1. **Connect to MongoDB:** First, establish a connection to the MongoDB database within the Flask application. This can be done by importing the `MongoClient` from PyMongo and creating a connection instance.

```
from flask import Flask

from pymongo import MongoClient

app = Flask(__name__)

mongo_client = MongoClient('mongodb://localhost:27017/')

db = mongo_client['commodity_database']

collection = db['commodity_prices']
```

2. **Retrieve Data from MongoDB:** Create functions within the Flask application to retrieve the required data from MongoDB. These functions can be called from the route handlers to fetch the data:

```
def get_all_commodities():

    commodities = list(collection.find())

    return commodities

def get_commodity_by_name(name):

    commodity = collection.find_one({'name': name})

    return commodity

def search_commodities(query):

    results = list(collection.find({'$text': {'$search':
query}}))

    return results
```

```
from flask import Flask
```

The `get_all_commodities` function retrieves all documents from the `commodity_prices` collection. The `get_commodity_by_name` function retrieves a single document based on the commodity name. The `search_commodities` function performs a text search in MongoDB using the provided query string.

```
main.py M X
main.py > ...
30
31 # Function to update the home page data if necessary
32 def getHomePage_clean():
33     last_update = db.metadata.find_one({"type": "home_update_clean"})
34     if not last_update or (datetime.datetime.now() - last_update['timestamp']).total_seconds() > 86400:
35         createHomePage_clean()
36         db.metadata.update_one(
37             {"type": "home_update_clean"},
38             {"$set": {"timestamp": datetime.datetime.now()}},
39             upsert=True
40         )
41
42 # Function to update the commodity data if necessary
43 def getCommodities(name, url):
44     last_update = db.metadata.find_one({"type": "commodity_update", "name": name})
45     if not last_update or (datetime.datetime.now() - last_update['timestamp']).total_seconds() > 86400:
46         createComDB(name, url)
47         db.metadata.update_one(
48             {"type": "commodity_update", "name": name},
49             {"$set": {"timestamp": datetime.datetime.now()}},
50             upsert=True
51         )
52
53
54 @app.route('/')
55 def home():
56     return render_template("index.html")
57
58
59 @app.route('/commodities')
60 def get_all_commodities(vis=False):
61     getHomePage()
62     commodities = db.commodities.find()
63     if vis:
64         return redirect(url_for("get_all_commodities_clean"))
65     return render_template("all_commodities.html", commodities=commodities)
66
```

Figure 3.7: Data retrieval from MongoDB

- **Pass Data to Templates:** In the route handlers, call the appropriate data retrieval functions and pass the retrieved data to the corresponding templates for rendering:

```
@app.route('/commodities')
```

```
def commodities():
```

```
    commodity_data = get_all_commodities()
```

```
    return render_template('commodities.html',
data=commodity_data)
```

```
@app.route('/commodity/<name>')
```

```
def commodity_details(name):
```

```
    commodity = get_commodity_by_name(name)
```

```
    return render_template('commodity_details.html',
commodity=commodity)
```

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

```
def search():
```

```
    if request.method == 'POST':
```

```
        query = request.form['search_query']
```

```
results = search_commodities(query)
```

```
return render_template('search_results.html',  
results=results)
```

```
return render_template('search.html')from flask  
import Flask
```

In the `commodities` route handler, the `get_all_commodities` function is called, and the retrieved data is passed to the `commodities.html` template for rendering. Similarly, in the `commodity_details` route handler, the `get_commodity_by_name` function is called with the commodity name from the URL parameter, and the retrieved document is passed to the `commodity_details.html` template. The `search` route handler handles the search form submission, calls the `search_commodities` function with the search query, and passes the search results to the `search_results.html` template.

4] RESULTS AND DISCUSSIONS

4.1 Data Scraping Performance Evaluation

Evaluating the performance of the web scraping process is crucial to ensure the quality, efficiency, and scalability of the data extraction pipeline. In this section, we will discuss two key aspects of performance evaluation: accuracy and completeness of scraped data, and efficiency and scalability considerations.

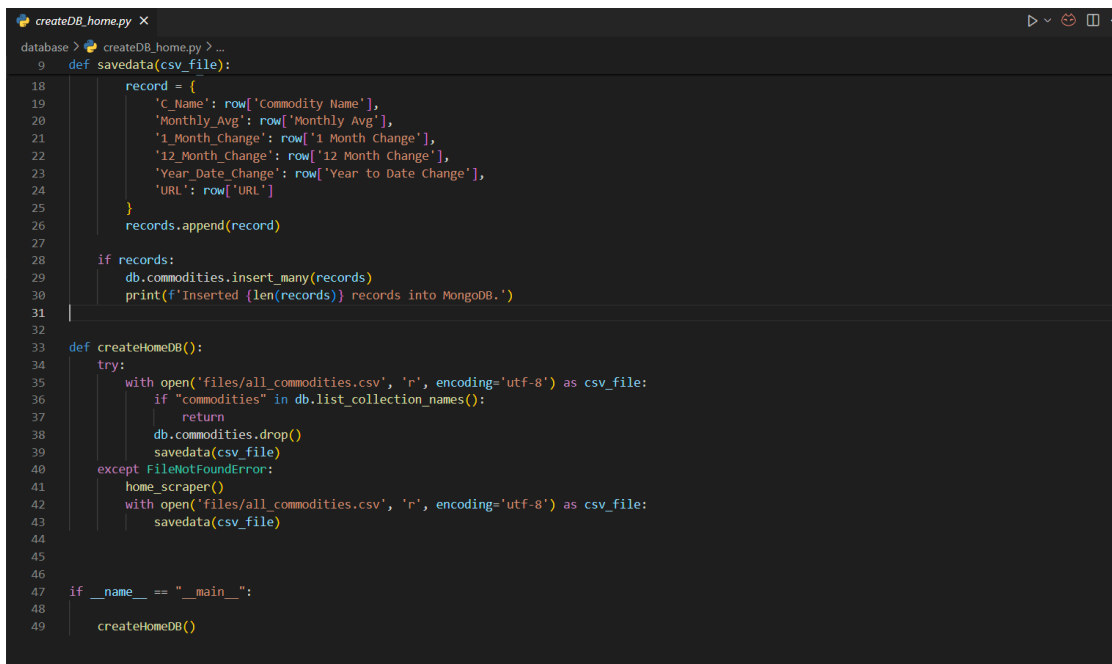
4.1.1 Accuracy and Completeness of Scraped Data

The accuracy and completeness of the scraped data are essential for ensuring the reliability and usefulness of the extracted information. Evaluating these aspects involves the following steps:

1. **Ground Truth Establishment:** Establish a ground truth dataset by manually verifying a representative sample of the data available on the target website (IndexMundi). This ground truth dataset will serve as a benchmark for evaluating the accuracy and completeness of the scraped data.
2. **Data Comparison:** Compare the scraped data with the ground truth dataset to identify any discrepancies or missing information. This can be done by:
 - **Value Comparison:** Compare the individual values (e.g., commodity prices, units, currencies) extracted by the scraper with the corresponding values in the ground truth dataset.
 - **Record Completeness:** Ensure that all relevant records (e.g., commodities, time periods) present in the ground truth dataset are captured by the scraper.
 - **Field Completeness:** Verify that all necessary fields or attributes (e.g., commodity name, price, date, source) are extracted for each record.
3. **Error Analysis:** Analyze the identified discrepancies or missing data to understand the root causes. This may involve:
 - **Website Structure Changes:** Monitor for any changes in the website's HTML structure or data representation that could impact the scraper's ability to accurately extract data.
 - **Data Format Variations:** Identify any variations in data formats or patterns that the scraper may not be handling correctly.
 - **Scraper Logic Flaws:** Examine the scraper's code for any logical errors or edge cases that might lead to inaccurate or incomplete data extraction.
4. **Performance Metrics:** Define and calculate relevant performance metrics to quantify the accuracy and completeness of the scraped data. Examples of metrics include:
 - **Data Accuracy Rate:** The percentage of scraped data values that match the ground truth dataset.
 - **Record Completeness Rate:** The percentage of records present in the ground truth dataset that are successfully captured by the scraper.
 - **Field Completeness Rate:** The percentage of required fields or attributes that are successfully extracted for each record.

5. **Iterative Improvement:** Based on the error analysis and performance metrics, identify areas for improvement and refine the scraper's logic, error handling, and data validation mechanisms to enhance the accuracy and completeness of the scraped data.

By regularly evaluating the accuracy and completeness of the scraped data, you can ensure the reliability and quality of the extracted information, which is crucial for downstream data analysis and decision-making processes.



```
createDB_home.py X
database > createDB_home.py > ...
9 def savedata(csv_file):
18     record = {
19         'C Name': row['Commodity Name'],
20         'Monthly_Avg': row['Monthly Avg'],
21         '1_Month_Change': row['1 Month Change'],
22         '12_Month_Change': row['12 Month Change'],
23         'Year_Date_Change': row['Year to Date Change'],
24         'URL': row['URL']
25     }
26     records.append(record)
27
28 if records:
29     db.commodities.insert_many(records)
30     print(f'Inserted {len(records)} records into MongoDB.')
31
32
33 def createHomeDB():
34     try:
35         with open('files/all_commodities.csv', 'r', encoding='utf-8') as csv_file:
36             if "commodities" in db.list_collection_names():
37                 return
38             db.commodities.drop()
39             savedata(csv_file)
40     except FileNotFoundError:
41         home_scraper()
42         with open('files/all_commodities.csv', 'r', encoding='utf-8') as csv_file:
43             savedata(csv_file)
44
45
46
47 if __name__ == "__main__":
48
49     createHomeDB()
```

Figure 4.1: Efficient data uploading

4.1.2 Efficiency and Scalability Considerations

In addition to accuracy and completeness, it is essential to evaluate the efficiency and scalability of the web scraping process, especially when dealing with large volumes of data or frequent scraping requirements.

1. **Scraping Performance Metrics:** Implement mechanisms to measure and monitor the performance of the web scraping process. Key metrics to consider include:
 - **Scraping Speed:** The rate at which data is extracted, measured in records per second or pages per second.
 - **Resource Utilization:** The CPU, memory, and network bandwidth consumption during the scraping process.
 - **Error Rates:** The frequency of errors or exceptions encountered during the scraping process.
2. **Load Testing:** Conduct load testing to evaluate the scraper's performance under different levels of concurrency and data volumes. This can help identify potential bottlenecks, resource constraints, or scalability issues.
3. **Caching and Parallelization Strategies:** Implement caching mechanisms to store and reuse previously scraped data, reducing the need for redundant

requests and improving overall efficiency. Additionally, explore parallelization techniques, such as multi-threading or distributed scraping, to leverage multiple CPU cores or machines for faster data extraction.

4. **Rate Limiting and Politeness Considerations:** Ensure that the scraper adheres to the target website's terms of service, rate limiting policies, and robot exclusion protocols (robots.txt). Implement mechanisms to regulate the scraping rate and avoid overloading the website with excessive requests, which could lead to IP blocking or legal consequences.
5. **Scalability Planning:** Evaluate the scalability requirements of the web scraping process based on factors such as the expected growth in data volume, frequency of scraping, and potential future use cases. Plan for scaling strategies, such as horizontal scaling (adding more machines) or vertical scaling (increasing resource allocation), to accommodate future growth.
6. **Monitoring and Alerting:** Implement monitoring and alerting mechanisms to proactively detect performance issues, errors, or deviations from expected behavior. This can help identify and address problems promptly, ensuring the continuous and reliable operation of the web scraping process.

By evaluating the efficiency and scalability of the web scraping process, you can optimize resource utilization, minimize downtime, and ensure that the data extraction pipeline can handle increasing data volumes and scraping requirements without compromising performance or reliability.

4.2 Analysis of Extracted Commodity Price History

After successfully scraping and storing the commodity price data in MongoDB, it is essential to analyze and derive insights from the extracted information. This section focuses on conducting statistical analysis, visualizing trends and patterns, and drawing meaningful observations from the commodity price history.



```
clean_homepage.py 1 X
analysis > clean_homepage.py > clean_data
1 import pandas as pd
2
3 def clean_data():
4     # read csv file
5     home_data = pd.read_csv('files/all_commodities.csv')
6
7     # condition to remove data
8     condition = home_data['1 Month Change'] == '\xa0'
9     df_filtered = home_data[~condition]
10
11     # save new csv
12     cleaned_file_path = 'files/cleaned_commodities.csv'
13     df_filtered.to_csv(cleaned_file_path, index=False)
```

Figure 4.2: Remove null values

4.2.1 Descriptive Statistics (Mean, Median, Standard Deviation)

Descriptive statistics provide a summary of the central tendency, dispersion, and distribution of the commodity price data. Calculating and reporting these statistics can offer valuable insights into the overall behavior and characteristics of the prices.

1. **Mean (Average):** Calculate the mean or average price for each commodity over a specified time period (e.g., monthly, quarterly, annually). The mean price can provide a general understanding of the typical price level for a commodity.
2. **Median:** Determine the median price for each commodity, which represents the middle value when the prices are arranged in ascending or descending order. The median can be a more robust measure of central tendency, as it is less influenced by outliers or extreme values.
3. **Standard Deviation:** Compute the standard deviation of prices for each commodity, which measures the spread or dispersion of the prices around the mean. A higher standard deviation indicates greater price volatility, while a lower standard deviation suggests more stable prices.
4. **Minimum and Maximum Values:** Identify the minimum and maximum prices observed for each commodity over the analyzed time period. These values can provide insights into the price range and potential price extremes.
5. **Quartiles and Percentiles:** Calculate quartiles (25th, 50th, and 75th percentiles) or other percentiles of interest to understand the distribution of prices and identify any potential outliers or anomalies.

By presenting these descriptive statistics, either in tabular or graphical form, stakeholders can gain a comprehensive understanding of the central tendencies, dispersion, and distribution of commodity prices, enabling informed decision-making and risk assessment.

4.2.2 Visualization of Trends and Patterns (Time Series Plots)

Visualizing the commodity price data over time can reveal valuable trends, patterns, and potential relationships that might not be immediately apparent from numerical data alone. Time series plots are particularly effective for this purpose.

1. **Line Plots:** Create line plots with time on the x-axis and commodity prices on the y-axis. These plots can effectively showcase the price movements and trends for individual commodities over time.
2. **Multiple Line Plots:** To facilitate comparisons, plot multiple commodity prices on the same chart, using different colors or line styles to distinguish between commodities. This can help identify potential correlations or divergences in price movements among different commodities.
3. **Interactive Visualizations:** Consider using interactive visualization tools or libraries (e.g., Plotly, D3.js) that allow users to zoom in/out, hover over data points for detailed information, and apply filters or selections to the data.
4. **Seasonality and Cyclicity:** Analyze the time series plots for any recurring patterns or seasonal fluctuations in commodity prices. These patterns can provide insights into market dynamics, supply and demand cycles, or other factors influencing price movements.

5. **Trend Lines and Curve Fitting:** Overlay trend lines or apply curve fitting techniques (e.g., linear regression, polynomial regression) to the time series data to identify long-term trends or potential forecasting models.

By visualizing the commodity price data over time, stakeholders can gain a better understanding of historical trends, identify potential turning points or inflection points, and uncover relationships or correlations between different commodities or market factors.

4.2.3 Insights and Observations (Market Fluctuations, Price Correlations)

Based on the descriptive statistics and visualizations, it is essential to derive meaningful insights and observations from the commodity price history. This section focuses on analyzing market fluctuations, identifying potential price correlations, and drawing other relevant conclusions.

1. **Market Fluctuations:** Examine the time series plots and descriptive statistics to identify periods of significant price fluctuations or volatility for specific commodities. Investigate potential causes or events (e.g., supply disruptions, geopolitical factors, economic conditions) that may have contributed to these fluctuations.
2. **Price Correlations:** Analyze the time series plots and descriptive statistics to identify potential correlations or relationships between the prices of different commodities. For example, certain commodities may exhibit similar price movements due to shared market factors or interconnected supply chains.
3. **Supply and Demand Factors:** Investigate the impact of supply and demand factors on commodity prices. Factors such as production levels, inventory levels, consumption patterns, and global trade dynamics can influence price movements and should be considered in the analysis.
4. **Macroeconomic Influences:** Explore the potential effects of macroeconomic factors, such as inflation, interest rates, exchange rates, and geopolitical events, on commodity price fluctuations. These factors can have far-reaching impacts on global markets and commodity prices.
5. **Industry-specific Factors:** For specific commodities or sectors, identify and analyze industry-specific factors that may influence price movements. These could include technological advancements, regulatory changes, or shifts in consumer preferences.
6. **Anomaly Detection:** Identify any anomalies or outliers in the price data that deviate significantly from expected patterns or historical norms. Investigate potential causes or data quality issues that may have contributed to these anomalies.
7. **Forecasting Potential:** Based on the identified trends, patterns, and relationships, explore the potential for developing forecasting models or techniques to predict future commodity price movements. However, it is important to acknowledge the inherent uncertainties and limitations of such forecasts.

By conducting a comprehensive analysis of the extracted commodity price history, stakeholders can gain valuable insights into market dynamics, identify potential risks

and opportunities, and make more informed decisions related to commodity trading, risk management, or investment strategies.

4.3 Website Functionality Demonstration

To showcase the practical application of the web scraping project and the integration of the extracted data into a user-friendly interface, this section focuses on demonstrating the functionality of the developed Flask web application.



Copyright © Your Website 2024

Figure 4.3: Homepage

4.3.1 User Interface Screenshots

Providing visual representations of the web application's user interface can effectively communicate the application's features and user experience. Include screenshots of the following aspects of the web application:

1. **Homepage:** Capture a screenshot of the web application's homepage, which typically serves as the entry point for users. Highlight any key features or navigation elements present on the homepage.

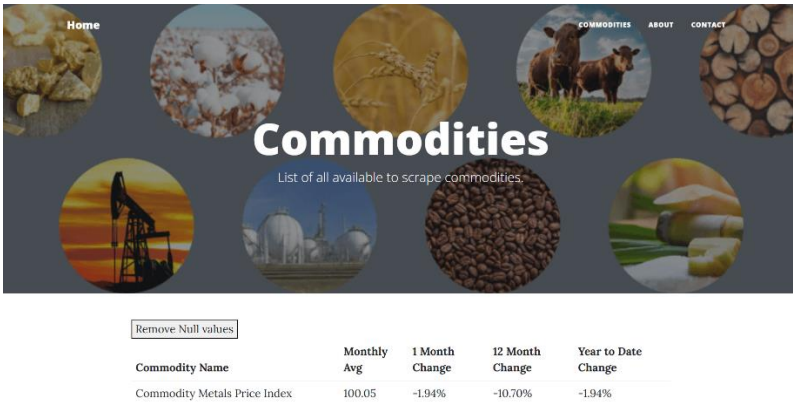


Figure 4.4: Commodities Homepage

2. **Commodity List View:** Include a screenshot of the view that displays a list or table of available commodities and their corresponding prices. This view should provide an overview of the scraped commodity price data.

Home	Commodity Agricultural Raw Materials Index	77.08	0.85%	1.88%	0.85%
	Commodity Food Price Index	116.57	-1.25%	-11.03%	-1.25%
	Crude Oil (petroleum); West Texas Intermediate	76.70	3.75%	-0.18%	3.75%
	Indonesian Liquefied Natural Gas	14.21	-0.91%	-22.86%	-0.91%
	Natural Gas	1.72	-45.91%	-27.73%	-45.91%
	Coffee, Other Mild Arabicas	4.61	3.13%	-8.89%	3.13%
	Sorghum				
	Bananas	1.57	-2.48%	-5.42%	-2.48%
	Fish (salmon)	10.27	0.59%	8.22%	0.59%
	Sugar, U.S. import price	0.92	4.55%	13.58%	4.55%
	Coconut Oil	1,171.58	3.63%	7.81%	3.63%
	Palm Kernel Oil	1,034.17	5.80%	-0.24%	5.80%
	Rapeseed Oil	962.71	-0.96%	-19.00%	-0.96%
	Fine Wool	921.18	-5.03%	-22.75%	-5.03%

Figure 4.5: All commodities in a table

3. **Commodity Detail View:** Capture a screenshot of the view that displays detailed information for a specific commodity, such as historical price data, charts or visualizations, and any other relevant details.



Figure 4.6: Specific Commodity View

4. **Search and Filter Functionality:** If the web application includes search or filter functionality, provide screenshots demonstrating these features in action. Highlight the user interface elements for entering search queries or applying filters.

5. **User Interaction Elements:** Include screenshots showcasing any user interaction elements, such as dropdowns, checkboxes, or sliders, that allow users to customize the displayed data or apply additional filters.
6. **Error or Information Messages:** If applicable, provide screenshots of any error or informational messages displayed by the web application in response to user actions or specific scenarios.

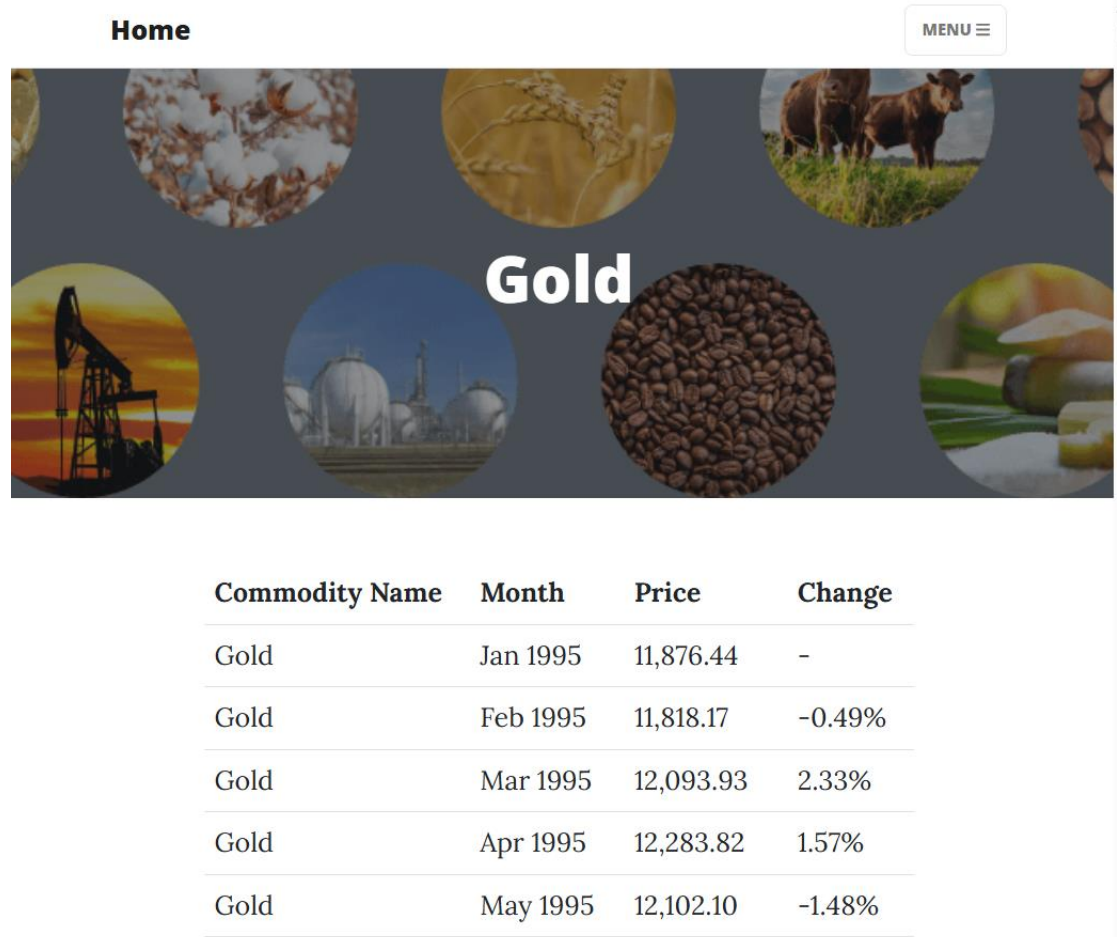


Figure 4.7: Dynamic Interaction with webpage

Accompany each screenshot with a brief description or caption explaining the purpose and functionality of the displayed user interface elements.

4.3.2 Explanations of Key Features and User Experience

In addition to the visual representations, provide detailed explanations of the key features and user experience offered by the web application. This section should cover the following aspects:

1. **Navigation and Information Architecture:** Explain the overall structure and organization of the web application's pages and views. Describe how users can navigate between different sections and access the desired information or functionalities.



Figure 4.8: Interactable navbar

2. **Data Presentation and Visualization:** Elaborate on the techniques and tools used for presenting the commodity price data in a user-friendly and visually appealing manner. Explain the rationale behind the chosen data visualization methods (e.g., tables, charts, graphs) and how they enhance user understanding.
3. **User Interaction and Customization:** Describe the various user interaction elements and customization options available within the web application. Explain how users can filter, search, or sort the displayed data based on their specific interests or requirements.
4. **Error Handling and Feedback Mechanisms:** Discuss the error handling mechanisms implemented in the web application. Explain how the application provides feedback or error messages to users in case of unexpected scenarios or user input errors.
5. **Responsive Design and Cross-device Compatibility:** If the web application is designed to be responsive and compatible across different devices (desktop, tablet, mobile), describe the techniques and frameworks used to ensure a consistent and optimized user experience across various screen sizes and resolutions.
6. **Accessibility Considerations:** If any accessibility features or considerations have been implemented in the web application, highlight them and explain how they enhance the user experience for individuals with disabilities or special needs.

5] Conclusion and Future Work

This section summarizes the achievements and contributions of the web scraping project, acknowledges its limitations and areas for improvement, and outlines potential future work directions to enhance and extend the project's capabilities.

5.1 Summary of Achievements and Contributions

The web scraping project successfully accomplished the following key achievements and contributions:

1. **Robust Web Scraping Pipeline:** A robust and scalable web scraping pipeline was developed using Python and the BeautifulSoup library. This pipeline efficiently extracted commodity price data from the IndexMundi website, ensuring accurate and complete data retrieval through comprehensive error handling and data validation mechanisms.

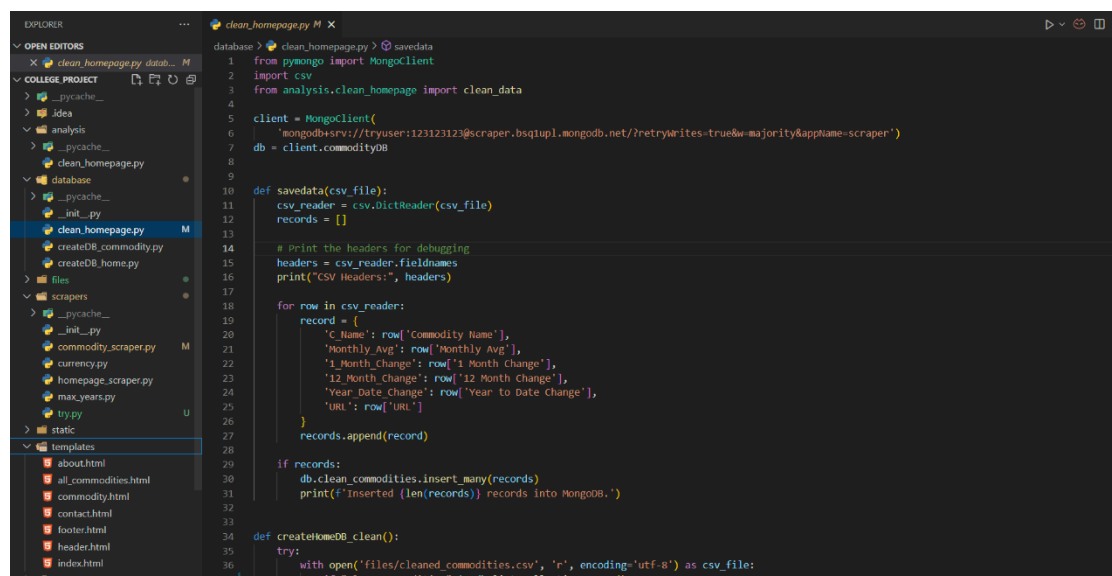


Figure 5.1: Pipeline and structure of app

2. **Flexible Data Storage:** The extracted commodity price data was seamlessly stored in a MongoDB database, leveraging its flexible document-based data model and scalability features. The database design and data transformation processes were carefully planned and implemented to ensure efficient data storage and retrieval.
3. **User-friendly Web Application:** A user-friendly web application was developed using the Flask web framework and the Bootstrap front-end framework. The application provided an intuitive and visually appealing interface for users to explore and interact with the commodity price data, including features such as data visualization, filtering, and search functionality.
4. **Data Analysis and Insights:** Comprehensive data analysis techniques were employed to derive valuable insights from the extracted commodity price

history. This included calculating descriptive statistics, visualizing trends and patterns using time series plots, and identifying potential market fluctuations, price correlations, and other relevant observations.

5. **Thorough Documentation and Reporting:** The project was thoroughly documented, including detailed explanations of the methodologies, implementation details, performance evaluations, and functionality demonstrations. This documentation serves as a valuable resource for future reference, collaboration, and potential extensions of the project.

```
_id: ObjectId('665d8e95987873973594de13')
C_Name: "Commodity Metals Price Index"
Monthly_Avg: "100.05"
1_Month_Change: "-1.94%"
12_Month_Change: "-10.70%"
Year_Date_Change: "-1.94%"
URL: "https://www.indexmundi.com/commodities/?commodity=metals-price-index"
```

```
_id: ObjectId('665d8e95987873973594de18')
C_Name: "Commodity Food and Beverage Price Index"
Monthly_Avg: "138.54"
1_Month_Change: "1.64%"
12_Month_Change: "-1.37%"
Year_Date_Change: "1.64%"
URL: "https://www.indexmundi.com/commodities/?commodity=food-and-beverage-pr..."
```

Figure 5.2: NoSQL database

Overall, the web scraping project demonstrated the successful integration of various technologies and techniques, including web scraping, data storage, web application development, and data analysis, to create a comprehensive solution for extracting, storing, and presenting commodity price data in a user-friendly and insightful manner.

5.2 Limitations and Areas for Improvement

While the web scraping project achieved its objectives, there are certain limitations and areas for improvement that should be acknowledged:

1. **Single Data Source:** The project currently relies on a single data source (IndexMundi) for commodity price data. Expanding the project to include additional data sources could provide a more comprehensive and diverse dataset, enhancing the breadth and reliability of the analysis.
2. **Limited Commodity Coverage:** The project focused on scraping and analyzing price data for a specific set of commodities. Expanding the scope to include a wider range of commodities, spanning various sectors and markets, could increase the project's relevance and applicability.
3. **Data Freshness and Update Frequency:** The current implementation assumes a static data source, where the scraped data remains relatively unchanged over time. Implementing mechanisms to handle dynamic data sources and regularly update the scraped data would enhance the project's ability to provide up-to-date and relevant information.
4. **Scalability Limitations:** While the project was designed with scalability in mind, as the volume of data and user traffic increases, additional optimizations and scaling strategies may be required to ensure consistent performance and responsiveness.
5. **Limited User Interaction and Customization:** While the web application provides basic user interaction and customization features, such as filtering and search, more advanced features like personalized dashboards, data export

options, and user-defined alerts or notifications could further enhance the user experience.

6. **Security and Access Control:** The current implementation does not include robust security measures or access control mechanisms. Incorporating features such as user authentication, role-based access control, and data encryption could enhance the project's security and enable more advanced use cases.

5.3 Future Work Directions

Based on the limitations and areas for improvement identified, several future work directions can be explored to enhance and extend the web scraping project:

Extending Functionality (Data Analysis Tools, User Authentication)

1. **Integrate Advanced Data Analysis Tools:** Explore the integration of more advanced data analysis tools and libraries, such as Pandas, NumPy, or scikit-learn, to enable more complex statistical analyses, data transformations, and predictive modeling capabilities.
2. **Implement User Authentication and Authorization:** Introduce user authentication and authorization mechanisms to enable personalized experiences, secure access to sensitive data, and support role-based permissions and access controls.
3. **Enhance User Interaction and Customization:** Implement additional user interaction and customization features, such as personalized dashboards, data export options, user-defined alerts or notifications, and customizable data visualization options.

Expanding Data Sources and Commodities

1. **Incorporate Additional Data Sources:** Extend the web scraping pipeline to include additional data sources, such as government agencies, industry reports, or specialized commodity trading platforms, to provide a more comprehensive and diverse dataset.
2. **Expand Commodity Coverage:** Broaden the scope of the project by scraping and analyzing price data for a wider range of commodities, spanning various sectors and markets, such as energy, metals, agriculture, and more.
3. **Implement Data Integration and Harmonization:** As the number of data sources and commodities increases, develop mechanisms for integrating and harmonizing data from multiple sources, ensuring consistency and enabling cross-source analysis and comparisons.

Integration with Machine Learning or Data Science Applications

1. **Develop Predictive Models:** Leverage the extracted commodity price data and integrate machine learning techniques to develop predictive models for

forecasting future price movements, identifying potential market trends, or detecting anomalies.

2. **Implement Sentiment Analysis:** Explore the integration of sentiment analysis techniques by analyzing news articles, social media data, or other textual sources related to commodities. This could provide additional insights into market sentiments and their potential impact on commodity prices.
3. **Build Recommendation Systems:** Develop recommendation systems that leverage the commodity price data and user preferences to suggest potential investment opportunities, risk management strategies, or commodity trading decisions.
4. **Incorporate Natural Language Processing (NLP):** Integrate NLP techniques to enable users to interact with the application using natural language queries, enhancing the user experience and accessibility of the commodity price data.

By exploring these future work directions, the web scraping project can evolve into a more comprehensive and powerful data-driven solution, providing valuable insights and decision-support capabilities to stakeholders in the commodity trading, investment, and risk management domains.

REFERENCES

Web Scraping:

1. "Web Scraping with Python" by Ryan Mitchell (O'Reilly Media, 2018)

This book provides a comprehensive guide to web scraping using Python and various libraries such as BeautifulSoup, Scrapy, and Selenium.

2. "Web Scraping with Python" by Katharine Jarmul and Richard Lawson (Packt Publishing, 2021)

Another excellent resource that covers web scraping techniques, data extraction, and data cleaning using Python and popular libraries.

3. "Web Scraping with Python: Collecting Data from the Modern Web" by Ryan Mitchell (Packt Publishing, 2015)

An older but still relevant book that focuses on web scraping techniques and handling various challenges such as JavaScript-rendered content and APIs.

Flask Web Development:

1. "Flask Web Development: Developing Web Applications with Python" by Miguel Grinberg (O'Reilly Media, 2018)

This book is considered one of the best resources for learning Flask and building web applications with Python. It covers essential Flask concepts, database integration, user authentication, and more.

2. "Flask by Example" by Gareth Dwyer (Packt Publishing, 2018)

This book takes a hands-on approach to learning Flask by building various real-world applications, covering topics like RESTful APIs, deployment, and testing.

3. "Flask Framework Cookbook" by Sourav Mukherjee (Packt Publishing, 2022)

A practical guide that covers a wide range of Flask topics, including templates, databases, caching, authentication, and deployment.

4. The official Flask documentation: <https://flask.palletsprojects.com/>

The Flask documentation is an excellent resource for learning Flask and understanding its features and capabilities.