

Automatic Web Scraping Framework for Health Care News Detection

Walid TAIB

March 22, 2024

Abstract

This report presents the development and the implementation of an automated web scraping framework designed for the purpose of collecting fake and real news related to healthcare, a key component of the NGI-Search funded project ([Herefanmi](#)). The primary objective of this framework is to gather a diverse dataset that will be used for training large language models focused on detecting fake news related to healthcare. The framework integrates advanced web scraping techniques to systematically gather data from various online sources

Keywords— Herefanmi

1 Introduction

The HeReFaNMi project, funded by NGI-Search, under that title. We're developing a large language model that can detect if health news is credible or not, but we need a dataset to do that. For that, we're building a framework, like a tool, that will automatically gather news from different sources on the internet. This way, we can collect a big dataset and use it to train our model

2 Objectives

This project aims to develop an automated framework to collect data from various websites and utilize it to train a machine learning model for detecting fake news. The framework will employ web scraping techniques to extract text and other relevant features from diverse online sources. The collected data will undergo an ETL (Extract, Transform, Load) process involving cleaning, formatting, and labeling to prepare it for machine learning. Subsequently, the prepared data will be used to train a model that can effectively distinguish between factual news and fake news.

The Figure 1 shows a diagram of a retrieval augmented system (RAS). A RAS system combines an information retrieval component with a text classification model to improve the accuracy and reliability of the generated text.

The diagram shows:

- **Scrapers:** Scrapers collect data from external sources, such as websites and RSS feeds.
- **Source:** Sources are the external repositories of data from which the scrapers collect data, mainly website pages and RSS feeds for near real-time data streams. **ETL:** ETL stands for Extract, Transform, and Load. ETL systems are used to clean, process, and load data into a database. **Database:** The database stores the data collected by the scrapers and processed by the ETL system.
- **Vector DB/Knowledge Graph:** The vector DB/knowledge graph stores the data in a vectorized format, which makes it easy for the retrieval algorithms to search and retrieve data.
- **Embedding:** Embedding is a technique that converts text into vectors. The embedding layer is used to convert the text in the database into vectors
- **Generator:** The generator is a text generation model that is used to generate text based on the vectors from the embedding layer and the retrieved data from the vector DB/knowledge graph.

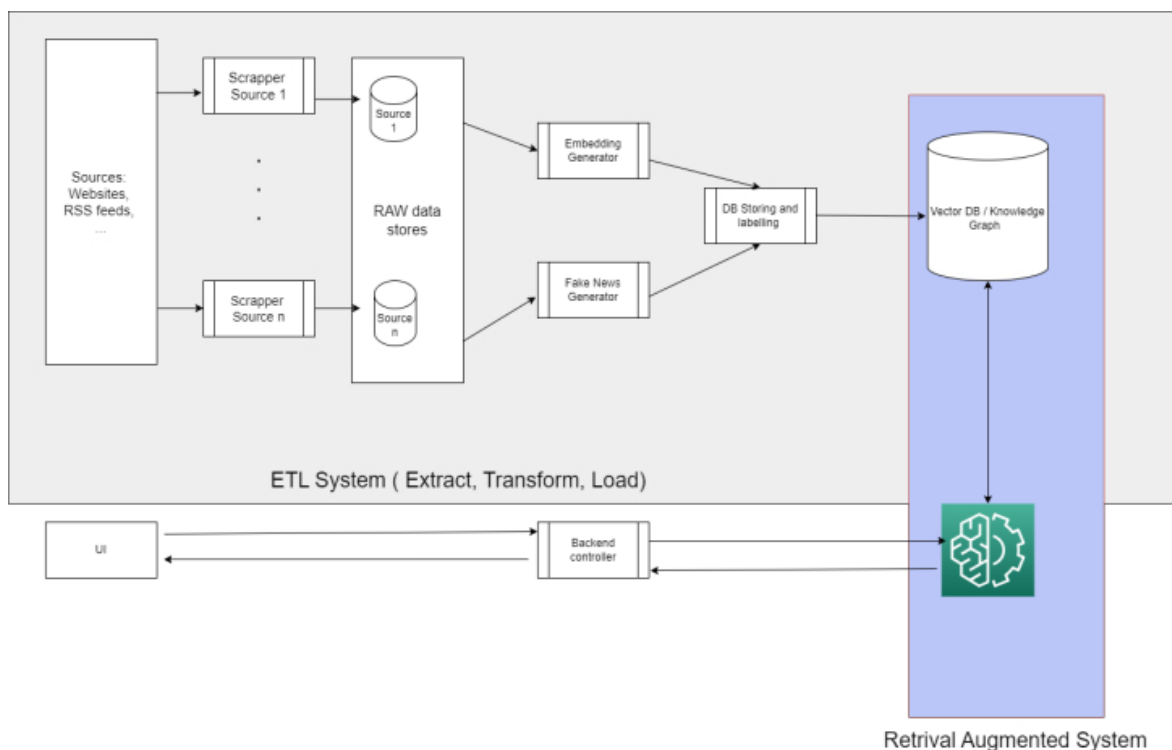


Figure 1:

- UI: The UI is the user interface of the RAS. It allows users to interact with the RAS and query it for information.
- Backend controller: The backend controller is responsible for managing the RAS and coordinating the interactions between the different components.

3 Accomplishments

The first week I did online presentation about fake news related to healthcare, where I explained the dynamic of fake news spread and why people believe it, I provided some relevant case studies with some statistics, also I listed a list of websites that spread real news and other websites that spread fake news related to healthcare.

Then, I wrote our first scraping script for [MedPageToday](#) a worth-trust platform posts daily news related to healthcare, and with that script, we were able to collect more than 25,000 items from 2005 until the day of its execution. In that script, I have used python and beautiful [Beautifulsoup](#) library.

And of-course some websites have so much data and the speed of scarping could take a long time I tried to integrate Thread-based parallelism by using [threading](#) library.

After that, I developed a simple interface with [ReactJs](#) and I have linked it with [flask](#) as a server-side to be able to execute the scraping script using REST-API to handle the request and the response between the front-end and the backend.

To make the framework more automatic I added some button to be able to upload and add new sources to the framework.

After finishing the process of collecting data I've started working on preprocessing or cleaning data for that I developed a script using different libraries of python to remove special characters, punctuation and handle missing values.

And then for the visualization I developed a script that does different types of visualizations commonly used in Natural Language Processing (NLP) to analyze text data such as [Word Clouds](#), [Heatmaps](#), Bar Charts and Histograms Scatter Plots.. and we integrated all this in our framework by adding more buttons for visualization and a simple interface to display the outputs.

4 Framework documentation

4.1 The architecture

This framework seeks to address the dimensions of data management, volume, value, variety, velocity, and veracity, so basically the framework is designed to efficiently collect data from diverse websites and ensuring that it is always up-to-date. It achieves this by facilitating the addition of new online data sources and by implementing mechanisms to scrape the new data (posted by the online data sources) that are available in our database.

The figure 2 is about the main architecture of the web scraping framework, it shows the following components:

- Scraper source: to extract data from a specific online source whether it is a website or RSS feeds.
- Router: manage the execution of different scraper sources based on the name received from the controller, so each website stored on the database is associated with a specific scraper source.
- controller: is the main or the root component where the overall structure of the application is defined. It is responsible to handle request from the interface, manage the database and the dataset....
- Preprocessing the data: to clean the data and to prepare it the training phase.
- Interface: Mainly the interface displays all the online data sources available on the data base, with buttons to be able to add new ones and to start scraping, as well as another button to visualize this data.

4.2 Code Implementation

- The interface is developed using ReactJS, the React folder composes from two parts :

First, we have App.js (see Listing 6), responsible for declaring the components. In our case, we currently have only one component, which is Main.js. However, we may need to create more components for more scalability.

```
1 //declare the Main component and for some styling bootstrap library
2
3 import Main from './components/Main';
4 import 'bootstrap/dist/css/bootstrap.min.css';
5
6 // The app function to use the main component
7
8 function App() {
9   return (
10     <div >
11       <Main/>
12     </div>
13   );
14 }
15
16 export default App;
17
18 }
```

Listing 1: Snipt code from App.js file

Then, Main.js (see Listing 2), responsible for retrieving the website's (online data source) information including the name, href and its type, using ("/api/get-data") API to display them in a table format and it has also three buttons for adding new online data source ("/api/post-data"), starting the scraping process ("/api/scrape-and-save") and another one for the visualization.

```
1
2 //This api is used to add new online website by including the href
3 and its type (credible or not)
4
```

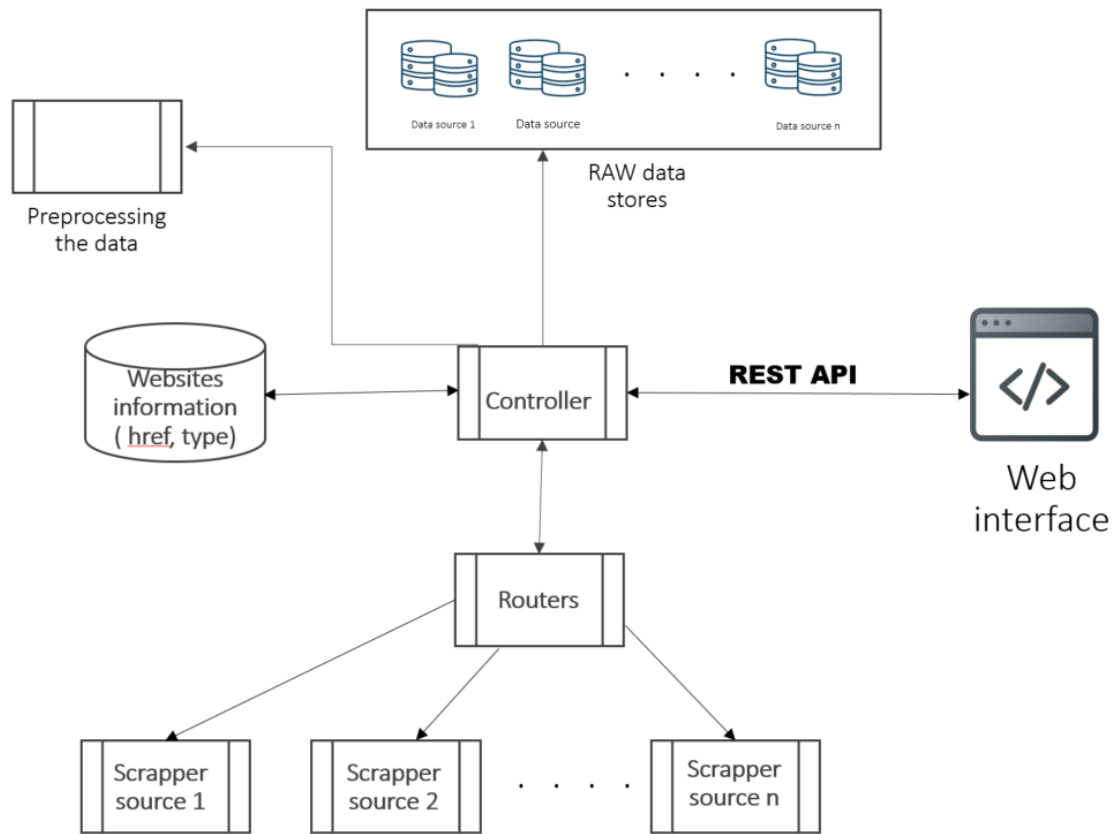


Figure 2: Automatic web scraping framework.

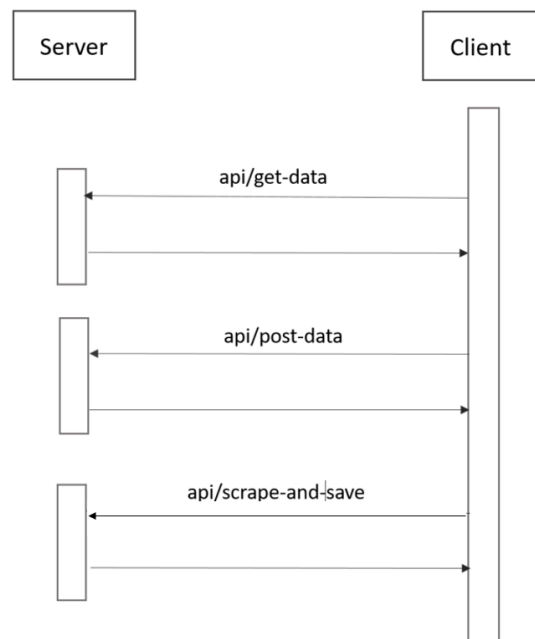


Figure 3: sequential diagram.

```

5  const handle_add_new_website = async (e) => {
6      e.preventDefault();
7
8      try {
9          await axios.post('http://localhost:5000/post_data', newData);
10         fetchData();
11         handleCloseModal();
12     } catch (error) {
13         console.error('Error submitting data:', error);
14     }
15 };
16
17 //This api is to start the scraping process
18
19 const handle_the_scraping = async (url) => {
20     console.log(url);
21
22     try {
23         setLoading(true);
24
25         await axios.post('http://localhost:5000/api/scrape_and_save', { url });
26
27         setLoading(false);
28     } catch (error) {
29         console.error('Error sending URL:', error);
30         setLoading(false);
31     }
32 };
33
34 //this api is to upload new scraper source
35
36 const handle_source_scraper_upload = async (item, file) => {
37     try {
38         if (file) {
39             const formData = new FormData();
40
41             const cleanFileName=item.url.replace(/https?:\/\//|www\.\|\.\w{2,3}/g,'').
replace(/\/g, '');
42
43             const newFileName = `${cleanFileName}.${file.name.split('.').pop()}`;
44
45             formData.append('file', file, newFileName);
46
47             const response = await axios.post('http://localhost:5000/upload_file',
formData, {
48                 headers: {
49                     'Content-Type': 'multipart/form-data',
50                 },
51             });
52
53             console.log('Response from server:', response.data);
54             fetchData();
55         }
56     } catch (error) {
57         console.error('Error uploading file:', error);
58     }
59 };

```

Listing 2: Snipt code from Main.js file

- The server-side developed using [flask](#), it has four main components:

First we have the Controller (see Listing 3). It is the root of the framework responsible for : starting the server, the connection between the server and the database, handling the request coming from the interface and for managing the dataset (storing and preprocessing)

```

1  #starting the flask server and define its requirements
2  app = Flask(__name__)
3  CORS(app)
4

```

```

5 app = Flask(__name__)
6 CORS(app)
7
8 # Connect to MongoDB
9 client = MongoClient('mongodb://localhost:27017/')
10 db = client['herefanmi_data_base']
11 collection = db['herefanmi_collection']
12
13
14 # display all the online sources using this api
15 @app.route('/get_data', methods=['GET'])
16 def get_data():
17     data = list(collection.find({}, {'_id': 0}))
18     return jsonify({'data': data})
19
20
21 # Add new source api
22 @app.route('/post_data', methods=['POST'])
23 def post_data():
24     try:
25         data_to_insert = request.get_json()
26         collection.insert_one(data_to_insert)
27         return jsonify({'message': 'Data successfully inserted!'})
28     except Exception as e:
29         return jsonify({'error': str(e)})
30
31
32 # upload new source api
33
34 @app.route('/upload_file', methods=['POST'])
35 def upload_file():
36     try:
37         if not os.path.exists(app.config['UPLOAD_FOLDER']):
38             os.makedirs(app.config['UPLOAD_FOLDER'])
39
40         uploaded_file = request.files['file']
41         clean_filename = secure_filename(uploaded_file.filename)
42
43         folder_name = os.path.splitext(clean_filename)[0]
44         print(folder_name)
45         folder_path = os.path.join(app.config['UPLOAD_FOLDER'], folder_name)
46
47         if not os.path.exists(folder_path):
48             os.makedirs(folder_path)
49
50         new_filename = os.path.join(folder_path, 'main.py')
51         uploaded_file.save(new_filename)
52
53         # Update MongoDB collection
54         url = f"www.{clean_filename.replace('.py', '')}.com"
55         print(url)
56         collection.update_one({'url': url}, {'$set': {'hasCodeSource': True}})
57
58         return jsonify({'message': 'File successfully uploaded and saved!', 'filename':
59 : clean_filename})
60     except Exception as e:
61         return jsonify({'error': str(e)})
62
63 if __name__ == '__main__':
64     from waitress import serve
65     serve(app, host='0.0.0.0', port=5000)

```

Snipet code from Controller.py file

Then we have the router (see Listing 3) to handle the execution of the scraper source according to the variable (script-path)

```

1
2 def execute_main_script(script_path, message_queue):
3     try:
4         # Execute the script
5         exec(open(script_path).read())
6     except Exception as e:

```

```

7         print(f"An error occurred while running the script: {e}")
8     finally:
9         # Send a message to the queue indicating completion
10        message_queue.put("COMPLETED")

```

Snipt code from router.py file

Finally, we have the scraper source 5, we used BeautifulSoup library from python and threading Thread-based parallelism to make the process more scalable and faster. In the scraper source we declare a class WebScraper, this class has two main functions scrape-urls to scrape the whole urls of the items of website and another function scrape-data-and-save to scrape the data from those urls.

```

1
2
3 #declare the webscraper class
4
5 class WebScraper:
6     def __init__(self):
7         self.base_url = 'https://www.medpagetoday.com/latest/np_{}'
8         self.second_url = 'https://www.medpagetoday.com/latest'
9         self.filename = 'website_data.json'
10        self.links = set() # Use a set to store unique links
11        self.is_beginning=True
12        self.lock = threading.Lock() # Create a lock to control access to shared
13        resources
14
15 #this function responisbles for scraping the urls from that online source
16 def scrape_urls(self, start_counter, thread_num, value):
17     response = requests.get(self.second_url)
18
19     if response.status_code == 200:
20         soup = BeautifulSoup(response.content, 'html.parser')
21
22         article_titles = soup.find_all('div', class_='article-title')
23         article_titles = article_titles[:21]
24
25         for title in article_titles:
26             anchor_tag = title.find('a')
27             if anchor_tag:
28                 href = anchor_tag.get('href')
29                 print('First link:', href)
30                 with self.lock:
31                     self.links.add(href) # Use add() to add unique URLs to the
32             set
33         else:
34             print('Failed to fetch the URL:', response.status_code)
35
36 # scrape and save function responsables for scraping data from the urls using threads
37
38 def scrape_data_and_save(url, csv_writer):
39     full_url = f"https://www.medpagetoday.com{url}"
40     print(full_url)
41     response = requests.get(full_url)
42
43     if response.status_code == 200:
44         html_content = response.content
45         soup = BeautifulSoup(html_content, "html.parser")
46
47         headline = soup.find("h1", class_="mpt-content-headline")
48         title = headline.text.strip() if headline else "Title not found"
49
50         h2_tag = soup.find('h2', class_='mpt-content-deck')
51         text = h2_tag.text.strip() if h2_tag else "Description not found"
52
53         meta_tag = soup.find('meta', attrs={'name': 'sailthru.date'})
54         date = meta_tag.get('content') if meta_tag else "N/A"
55
56         p_tags = soup.find_all('p')
57

```

```

58     # Concatenate text from all <p> tags into a single paragraph
59     concatenated_paragraph = '\n'.join(p.get_text(strip=True) for p in p_tags)
60
61     csv_writer.writerow([title, text, date, concatenated_paragraph]) # Write data
62     to CSV
63     print(f"Scraped data from {url}")
64 else:
    print(f"Failed to scrape data from {url}")

```

Snippet code from `scraper.py` file

4.3 Code documentation

Here's a documentation breakdown of the Frontend

4.3.1 Frontend Documentation:

Component Name: `DataDisplay`

This React component is responsible for displaying and managing data related to scraper sources. It fetches data from a backend API ('http://localhost:5000/getdata'), displays it in a table, and allows users to perform various actions on the data.

A Functionality:

- Fetches data on component mount using 'useEffect' hook.
- Displays data in a table with columns for ID, URL, Type, Code Source, and Actions.
- Shows a "Scraping..." message or a "Start Scraping" button based on the loading state for each URL.
- Allows uploading a file for sources without code source.
- Provides a modal for adding new scraper sources with URL, Type, and optional file upload.

B External Libraries:

- `React`: Used for building the UI components, providing a declarative and performant way to define the component's structure and behavior.
- `axios`: Used for making HTTP requests to the backend API, enabling communication with the server to fetch and update data.
- `react-bootstrap`: Used for creating the table, modal, buttons, and form elements, providing pre-built and styled UI components.

C Code Breakdown:

a) Imports:

- Imports necessary libraries (`React`, `useState`, `useEffect`, `axios`, components from `react-bootstrap`).

b) Data State:

- Uses `useState` hooks to manage various data:
 - `data`: Stores the fetched data from the API (an array of objects).
 - `showModal`: Controls the visibility of the "Add new Online Source" modal.
 - `newData`: Stores user input for adding a new source (URL, Type, and file).
 - `loadingStates`: Tracks the loading state (scraping) for each URL in the data.

c) Fetching Data:

- The `fetchData` function is called on component mount using `useEffect`.

- It makes a GET request to the backend API (`http://localhost:5000/getdata`) to retrieve scraper source data.
- On successful response, updates the `data` state with the fetched data.

d) Modal Handling:

- `handleShowModal` function sets `showModal` state to `true` to show the modal.
- `handleCloseModal` function sets `showModal` state to `false` to hide the modal.

e) User Input Handling:

- `handleInputChange` function updates the `newData` state based on user input in the modal form.
 - It uses the spread operator (`...prevData`) to create a new object with updated values.

f) File Upload Handling:

- `handleFileChange` function updates the `newData` state with the selected file from the modal form.

g) Adding New Source:

- `handleSubmit` function handles form submission in the modal.
 - It makes a POST request to the backend API (`http://localhost:5000/post-data`) with the new source data (`newData`).
 - On success, fetches data again (`fetchData`), closes the modal (`handleCloseModal`).

h) Scraping URL:

- `handleButtonClick` function takes a URL as input.
 - It updates the `loadingStates` for that URL to `true` to indicate scraping in progress.
 - Makes a POST request to the backend API (`http://localhost:5000/api/scrape-and-save`) with the URL.
 - On success, updates the `loadingStates` back to `false`.

i) Uploading File for Source:

- `handleFileUpload` function takes a source item and a file as input.
 - It constructs a `FormData` object with the file and a cleaned filename.
 - Makes a POST request to the backend API (`http://localhost:5000/upload-file`) with the `FormData`.
 - On success, fetches data again (`fetchData`).

4.3.4 Server-side Documentation:

As we mentioned earlier, there are three main components: Controller, Router and the scrappers

A External Libraries:

- `Flask`: Web framework for building web applications.
- `pymongo`: Library for interacting with MongoDB databases.
- `Flask-CORS`: Enables Cross-Origin Resource Sharing (CORS) for requests from different domains.
- `werkzeug`: Utility library for web development in Python, offering essential tools like routing, request/response handling, debugging

- Flask-CORS: Enables Cross-Origin Resource Sharing (CORS) for requests from different domains.
- urllib.parse: Parses URLs.
- threading: Enables running code concurrently in separate threads.
- json: Handles JSON data format.
- requests: Makes HTTP requests to external URLs.
- csv: Processes CSV files.
- bs4: Beautiful Soup library for parsing HTML content.
- queue: Provides thread-safe queues for communication

B MongoDB Connection:

The application connects to a MongoDB database named 'herefanmi-data' on the local machine (localhost) at port 27017. It interacts with a collection named 'herefanmi-collection' within this database.

C File Uploads

The application configures an upload folder using the 'UPLOAD-SCRAPPER' variable defined in the 'components.config' module. Uploaded files are saved within this folder.

4.4 Apis

The API provides several functionalities through different endpoints:

4.4.1 GET /getdata:

- Retrieves all data entries from the MongoDB collection.
- Response: JSON object containing a list of data entries, excluding the '-id' field.

4.4.2 POST /postdata:

- Inserts a new data entry into the MongoDB collection.
- Request Body: JSON object representing the data to be inserted.
- Response: JSON object with a success message upon successful insertion or an error message with details in case of failure.

4.4.3 POST /uploadfile:

- Uploads a Python script file containing scraping logic.
- Request Body: Multipart form data containing a file named 'file'.
- Response: JSON object with a success message, filename of the uploaded file, or an error message with details in case of failure.
- The uploaded file is saved within a folder named after the script filename (excluding the '.py' extension).
- The corresponding URL in the database collection (constructed as 'www.filename.replace('.py', '.com') is marked as having scraping code ('hasCodeSource' field set to 'True').

4.4.4 POST /api/scrape-and-save:

- Initiates the scraping process using the uploaded script for a provided URL.
- Request Body: JSON object containing a 'url' field specifying the target URL for scraping.
- Response: JSON object with a success message upon successful scraping and data saving or an error message with details in case of failure.
- The script is expected to be located within the uploads folder, following the same folder structure as created during upload.
- The scraping process runs in a separate thread using the 'execute-main-scripts' function (defined elsewhere).
- After scraping finishes, a cleaning function ('clean-csv') is applied to the generated CSV file ('output-data.csv') within the script's folder. The cleaned data is saved as 'output-cleaned.csv'.

4.5 Deployment avec Netlify

4.5.1 Front-end

In this sector we'll demonstrate how to deploy a React application to Netlify server.

To deploy the latest build, run npm run build beforehand and install netlify-cli globally:

```
1  
2 install netlify-cli -g
```

Listing 6: Install netlify-cli global

In the project root directory, run netlify command. You might be prompted to grant access to Netlify CLI. Click Authorize:

Authorize Application

Netlify CLI is asking for permission to access Netlify on your behalf.

This app will be able to create and manage sites in your Netlify teams. You can revoke access at any time.

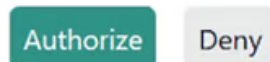


Figure 4: configuration of Netlify server step 01

Follow through the prompts that come after authentication. First, you'll be asked to link this project to a site, as this is the first time you're deploying this app. Select Create configure a new site:

```
This folder isn't linked to a site yet  
? What would you like to do?  
Link this directory to an existing site  
Create configure a new site
```

Figure 5: configuration of Netlify server step 02

Then, you'll be asked for the Team. Unless you're already using Netlify on your local machine, chances are you will see only one option with your name; select it:

```
? Team:
ASHUTOSH KUMAR SINGH's team
```

Figure 6: configuration of Netlify server step 03

The next prompt is Site name. This is an optional you can name it HeReFaNMi Web scraping framework

After this step, your site will be created and you'll be asked for a Publish directory. Type build here — the build directory will be created when you run `npm run build`:

```
Please provide a publish directory

? Publish directory build
```

Figure 7: configuration of Netlify server step 04

With this, your site will be published and you will be provided a draft URL:

Go to this draft URL. If everything checks out, you can deploy the website to the main site URL. Run the following command to deploy to production

```
netlify deploy --prod
```

Figure 8: configuration of Netlify server step 05

It will ask for the Publish directory one final time. Type build and hit Enter. You'll be provided with two URLs. The difference between these two URLs is that the Unique Deploy URL points to a specific version of your application. For example, if you make a change in your application and deploy again, you'll get another Unique Deploy URL that is specific to that change. Your Website URL is the main URL, which corresponds to the latest version of your application.

4.5.2 Backend with Google App Engine

You will need a Google Account. Once you create an account, go to the Google Cloud Platform Console and create a new project. Also, you need to install the Google Cloud SDK.

You will need to create three new files: `app.yaml`, `appengine-config.py`, and `requirements.txt`.

```
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /static
  static_dir: static
- url: /.*
  script: main.app

libraries:
- name: ssl
  version: latest
```

Figure 9: Content of `appengine-config.py`

Scraper Sources				
ID	URL	Type	Code Source	Action
<div> Add new Online Source </div>				

Figure 12: Add new online source step 01

The SSL Library allows us to create secure connections between the client and server.

```
from google.appengine.ext import vendor

# Add any libraries installed in the "lib" folder.
vendor.add('lib')
```

Figure 10: Content of requirements.txt

```
from google.appengine.ext import vendor

# Add any libraries installed in the "lib" folder.
vendor.add('lib')
```

Figure 11: Content of appengineconfig.py:

Now inside our virtual environment (make sure your virtualenv is activated), we are going to install the new dependencies we have in requirements.txt. Run this command: `pip install -t lib -r requirements.txt`

-t lib: This flag copies the libraries into a lib folder, which uploads to App Engine during deployment.

-r requirements.txt: Tells pip to install everything from requirements.txt

And finally to deploy the application to Google App Engine, use this command : `gcloud app deploy`

5 User Guide

5.1 Create new online data source

Go to the website, add your credential (email and password), then click on the button add new online source Figure 11

Then you will fill out the necessary information (href, and its type (credible or not)) as it is showing in the figure 13 and then just submit it.

5.2 Create New Scraper source code

First thing you need to do is to create scraper source with python language. So basically you need to declare the Webscraper class and the relevant functions: `scrape_urls` to scrape the urls of the items and save them in JSON file, `scrape_data` function to scrape the data using these urls.

For each source, scrape the following information:

- Title
- Description
- Details (if it is available)

Figure 13: Add new online source step 02

Scraper Sources

ID	URL	Type	Code Source	Action
	www.example.com	credible	✓	<button>Start Scraping</button>

Add new Online Source

Figure 14: Displaying the online sources

- Date

For the Requirements:

- The script should use the BeautifulSoup library for parsing HTML.
- The script should use the requests library to make HTTP requests to the website.
- The script should use these functions: scrape-urls to scrape the urls of the items and save them in JSON file scrape-data function to scrape the data using these urls and save them in a CSV format.
- The script should be efficient and able to scrape all items within a reasonable time frame we recommend to use threading library from python to distribute the urls between the threads.
- The script should handle errors gracefully and not crash if there are connection or parsing issues.

and upload the scrapper code to the framework, the online data source will displayed just like that [14](#)

5.3 Start the scrapping process

To start collecting data you need just to click on the button (Start scraping) [15](#) and it will start the scraping as it shows in the figure [16](#)

Scraper Sources



ID	URL	Type	Code Source	Action
	www.medpagetoday.com	credible	✓	 

Figure 15: Start the scraping

Scraper Sources

ID	URL	Type	Code Source	Action	Upload File
	www.example.com	credible	✓	Scraping...	



Figure 16: the scraping start