Documentation

WebScrapingAPI

Author: Zilai Denis

Contents

1. Introduction

This project is a simple web scraping application, which allows the users to retrieve the data from a website, in a JSON style text, while also including a feature to save the retrieved data as a .txt file.

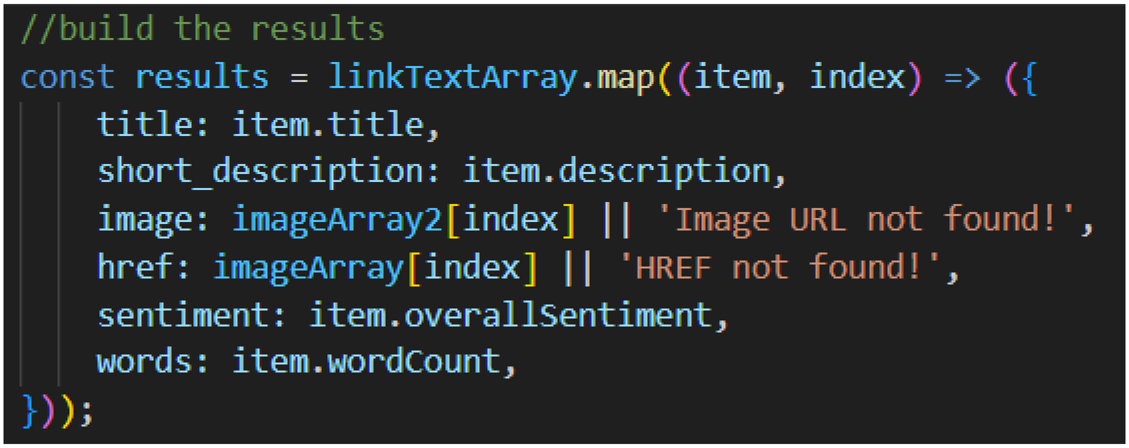
2. Implementation

2.1. Code overview

Backend

The code that the application has been written with is JavaScript, using the popular programming environment NodeJS. The structure of the code is fairly simple and clear, everything happening within one of the three endpoints used –will be elaborated later in this document--. The results of the API calls are delivered using JSON, allowing the user to read the data in a clear and well structured manner.

The structure of the returned results can be seen below:

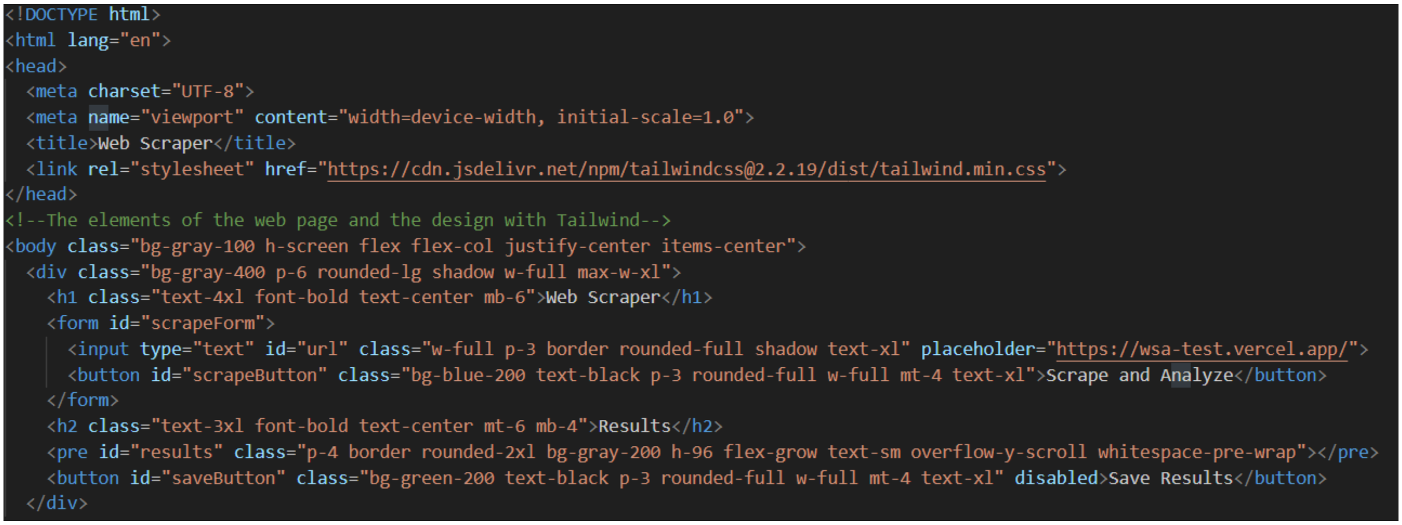


```
//build the results
const results = linkTextArray.map((item, index) => ({
    title: item.title,
    short_description: item.description,
    image: imageArray2[index] || 'Image URL not found!',
    href: imageArray[index] || 'HREF not found!',
    sentiment: item.overallSentiment,
    words: item.wordCount,
}));
```

Frontend

The frontend of the application was built using HTML, for defining the structure of the webpage. For styling, Tailwind has been used for the styling for the document, being an enhanced “version” of the well known CSS, as well as being lightweight and easy to use, producing better results.

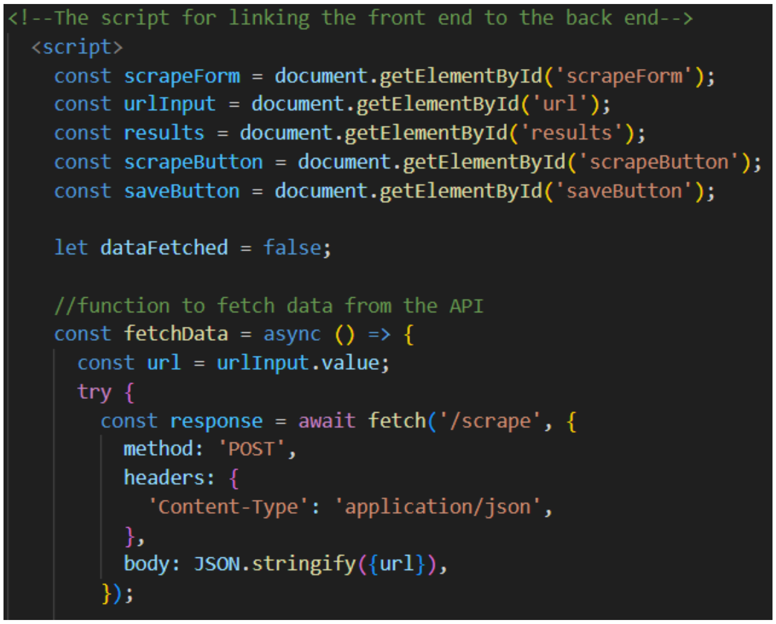
The structure of the HTML and Tailwind can be seen below:



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Web Scraper</title>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css">
</head>
<!--The elements of the web page and the design with Tailwind-->
<body class="bg-gray-100 h-screen flex flex-col justify-center items-center">
  <div class="bg-gray-400 p-6 rounded-lg shadow w-full max-w-xl">
    <h1 class="text-4xl font-bold text-center mb-6">Web Scraper</h1>
    <form id="scrapeForm">
      <input type="text" id="url" class="w-full p-3 border rounded-full shadow text-xl" placeholder="https://wsa-test.vercel.app/">
      <button id="scrapeButton" class="bg-blue-200 text-black p-3 rounded-full w-full mt-4 text-xl">Scrape and Analyze</button>
    </form>
    <h2 class="text-3xl font-bold text-center mt-6 mb-4">Results</h2>
    <pre id="results" class="p-4 border rounded-2xl bg-gray-200 h-96 flex-grow text-sm overflow-y-scroll whitespace-pre-wrap"></pre>
    <button id="saveButton" class="bg-green-200 text-black p-3 rounded-full w-full mt-4 text-xl" disabled>Save Results</button>
  </div>
```

The scripts for linking the frontend and the backend have been written in JavaScript, and can be found in the HTML document “index.html”.

A snippet of code:

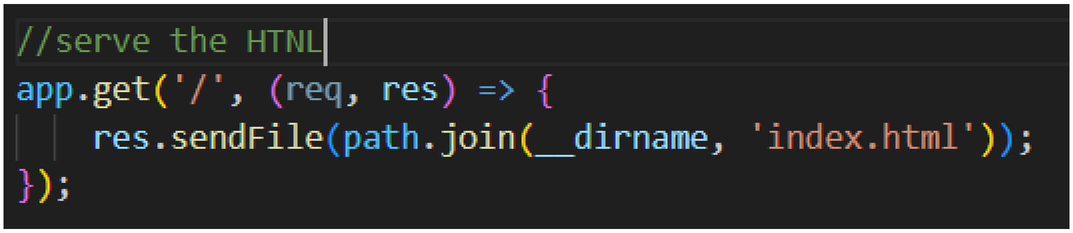


```
<!--The script for linking the front end to the back end-->
  <script>
    const scrapeForm = document.getElementById('scrapeForm');
    const urlInput = document.getElementById('url');
    const results = document.getElementById('results');
    const scrapeButton = document.getElementById('scrapeButton');
    const saveButton = document.getElementById('saveButton');

    let dataFetched = false;

    //function to fetch data from the API
    const fetchData = async () => {
      const url = urlInput.value;
      try {
        const response = await fetch('/scrape', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
          },
          body: JSON.stringify({url}),
        });
```

2.2 Endpoints

What needs to be accomplished happens between the defined endpoints. Next, the endpoints will be explained in a detailed manner.

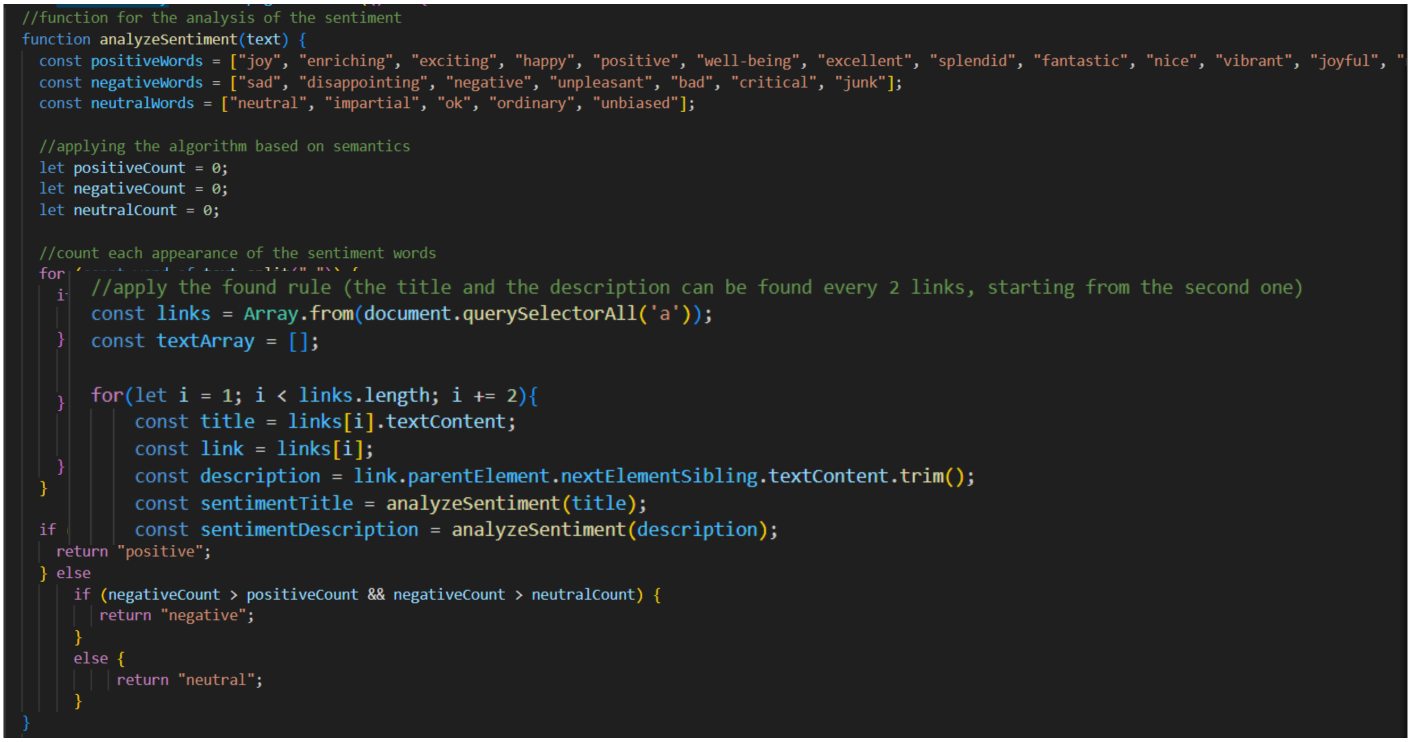
- '/' endpoint
  This endpoint serves the HTML content, with the help of the express module. When the user accesses the root URL 'http://localhost:3000/' in the web browser, the application responds by serving the HTML file.
  This endpoint can be visualized below:



```
//serve the HTNL
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});
```

- 'scrape' endpoint
  The main functionalities of this application are delivered within this endpoint.
  Firstly, the code within this endpoint is encapsulated in a try-catch-finally block, to catch errors that could appear during the scraping.
  The target website has been observed to be dynamically loading, so awaits have been used, to wait for the page to load its elements.
  Next, in the const linkTextArray, will be retrieved the title name, the short description, the overall sentiment and the word count. The function for determining the overall sentiment of a post from the website has been chosen as the best option for the task. The algorithm of this function is based

on lexicon-based sentiment analysis, where it is needed to determine the overall sentiment of a text, from the three decided. The most specific keywords to determine the sentiment have been added in arrays, and then counted the appearances of each with the text. The sentiment count with the most appearances will define the overall sentiment of the post. The function can be seen below:



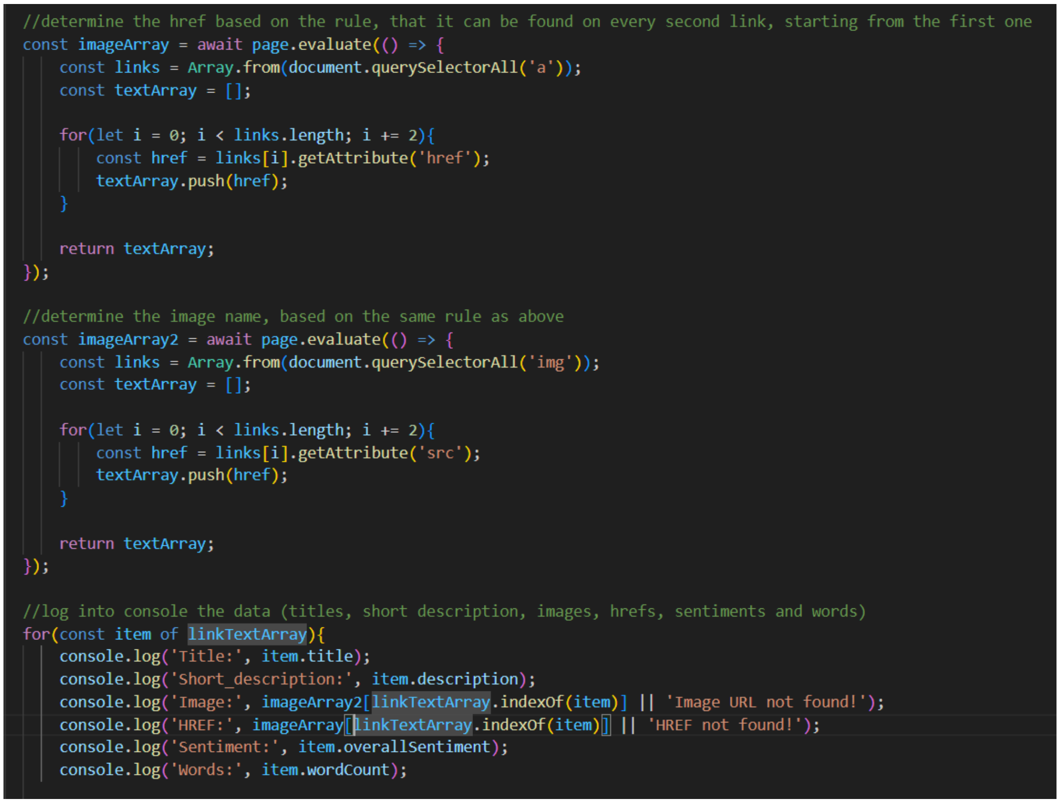
```
//function for the analysis of the sentiment
function analyzeSentiment(text) {
  const positiveWords = ["joy", "enriching", "exciting", "happy", "positive", "well-being", "excellent", "splendid", "fantastic", "nice", "vibrant", "joyful", "
  const negativeWords = ["sad", "disappointing", "negative", "unpleasant", "bad", "critical", "junk"];
  const neutralWords = ["neutral", "impartial", "ok", "ordinary", "unbiased"];

  //applying the algorithm based on semantics
  let positiveCount = 0;
  let negativeCount = 0;
  let neutralCount = 0;

  //count each appearance of the sentiment words
  for
    i
    }

    }

    }
  }

  if
    return "positive";
  } else
    if (negativeCount > positiveCount && negativeCount > neutralCount) {
      return "negative";
    }
    else {
      return "neutral";
    }
}
```

```
//apply the found rule (the title and the description can be found every 2 links, starting from the second one)
const links = Array.from(document.querySelectorAll('a'));
const textArray = [];

for(let i = 1; i < links.length; i += 2){
    const title = links[i].textContent;
    const link = links[i];
    const description = link.parentElement.nextElementSibling.textContent.trim();
    const sentimentTitle = analyzeSentiment(title);
    const sentimentDescription = analyzeSentiment(description);
```

Further, it was needed to determine the titles and the short descriptions. For these, has been found a rule, that being that the title and the short description are paired two by two, and can be found starting from the second link. This rule has been determined by analyzing the HTML source code of the provided web page. Within this process, the sentiment has been also determined for the titles and short descriptions.

For the next task, it was needed to count the words in the titles and descriptions. This was done by splitting the words, determining the length of the titles and descriptions, and then adding them. After that, the fetched data has been pushed in the textArray array, which will be returned and sent.

Going further, it was needed to find the HREF of the images from the web page. A rule was needed again, this time finding the necessary information with every second link, starting from the first one. The image name was also fetched using the same rule (from analyzing the HTML source code).



```
//determine the href based on the rule, that it can be found on every second link, starting from the first one
const imageArray = await page.evaluate(() => {
    const links = Array.from(document.querySelectorAll('a'));
    const textArray = [];

    for(let i = 0; i < links.length; i += 2){
        const href = links[i].getAttribute('href');
        textArray.push(href);
    }

    return textArray;
});

//determine the image name, based on the same rule as above
const imageArray2 = await page.evaluate(() => {
    const links = Array.from(document.querySelectorAll('img'));
    const textArray = [];

    for(let i = 0; i < links.length; i += 2){
        const href = links[i].getAttribute('src');
        textArray.push(href);
    }

    return textArray;
});

//log into console the data (titles, short description, images, hrefs, sentiments and words)
for(const item of linkTextArray){
    console.log('Title:', item.title);
    console.log('Short_description:', item.description);
    console.log('Image:', imageArray2[linkTextArray.indexOf(item)] || 'Image URL not found!');
    console.log('HREF:', imageArray[linkTextArray.indexOf(item)] || 'HREF not found!');
    console.log('Sentiment:', item.overallSentiment);
    console.log('Words:', item.wordCount);
```

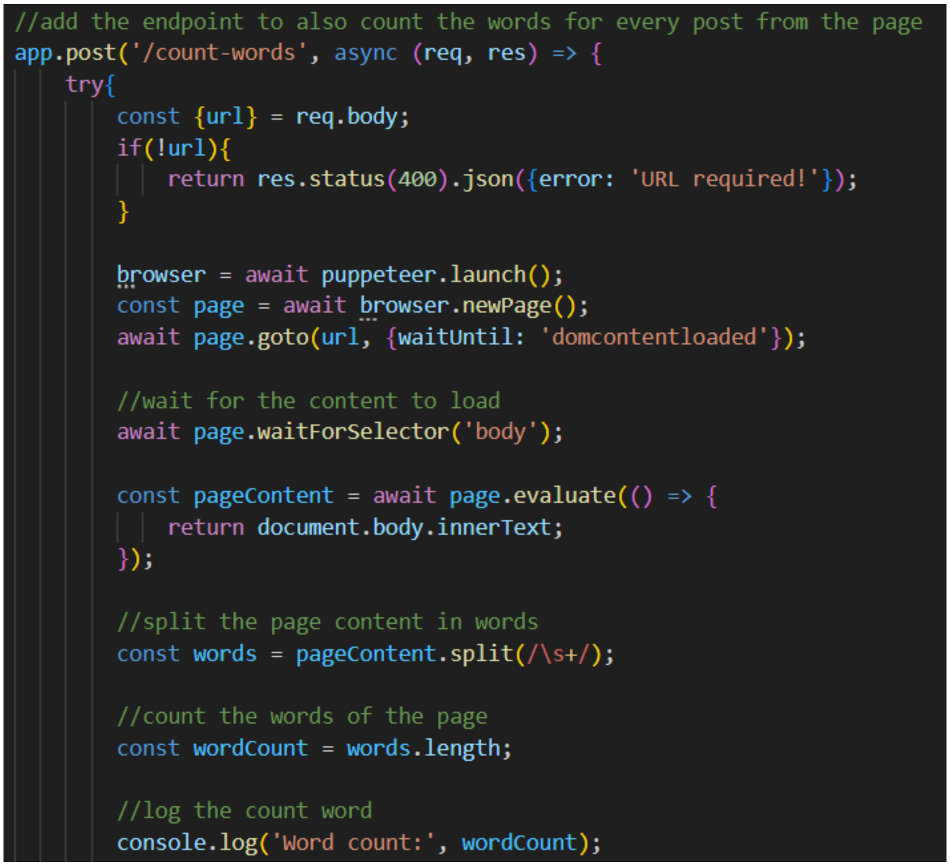
Finally, the fetched data was printed in the console log, as well as building the results, so that the final and complete batch of data to be sent as a JSON response.

- '/count-words' endpoint

  The use of this endpoint is to determine the words count from every post from the page (the posts can be accessed by clicking either the image or the post title).

  This endpoint is very similar to the previous one, the difference here being that the whole page has been split into words, and then determining the length of the constant "words", to find out the total number of words. This API call can be made by inputting the URL of the desired page in the API calls tool (for this project, **Insomnia** has been used).

Finally, for all of the endpoints, the application listens on the port 3000.



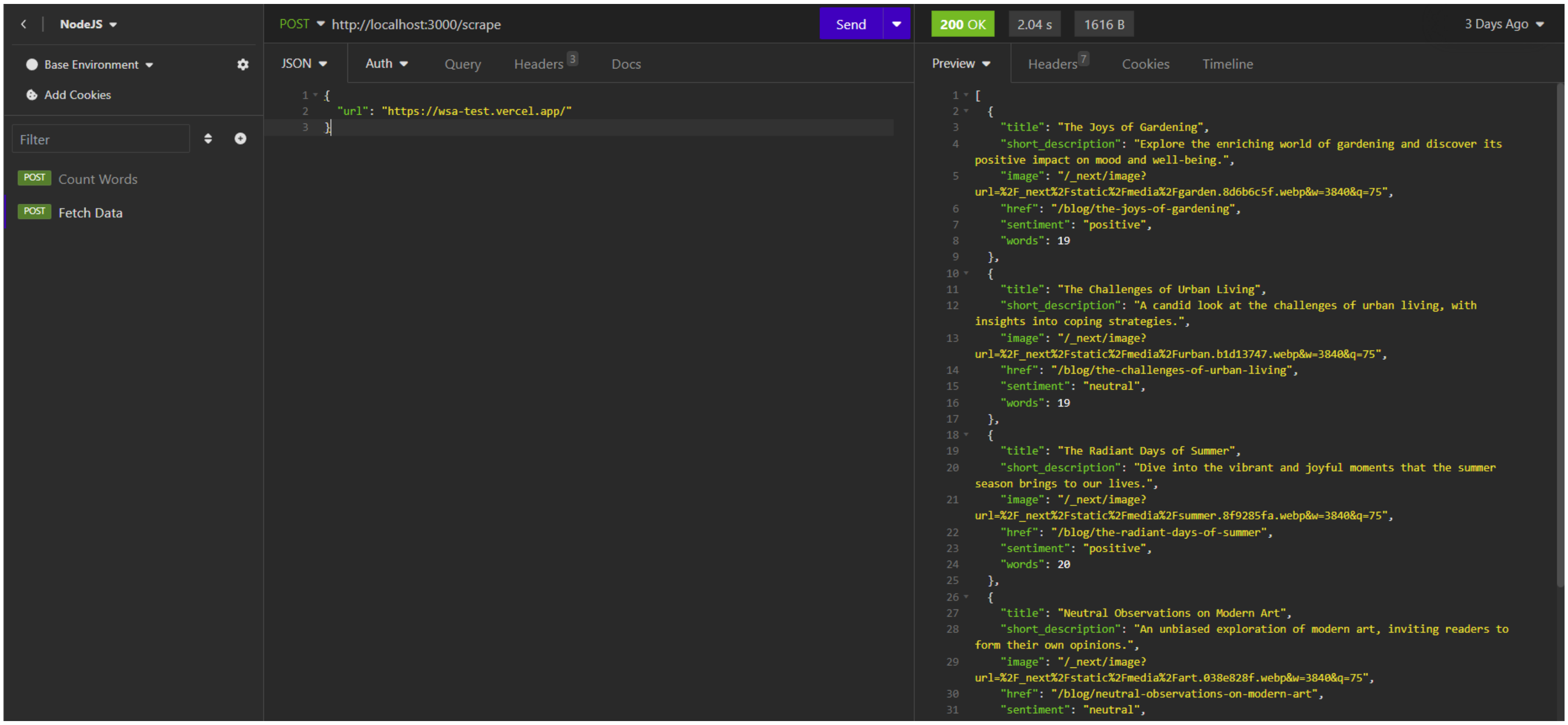
```
//add the endpoint to also count the words for every post from the page
app.post('/count-words', async (req, res) => {
    try{
        const {url} = req.body;
        if(!url){
            return res.status(400).json({error: 'URL required!'});
        }

        browser = await puppeteer.launch();
        const page = await browser.newPage();
        await page.goto(url, {waitUntil: 'domcontentloaded'});

        //wait for the content to load
        await page.waitForSelector('body');

        const pageContent = await page.evaluate(() => {
            return document.body.innerText;
        });

        //split the page content in words
        const words = pageContent.split(/\s+/);

        //count the words of the page
        const wordCount = words.length;

        //log the count word
        console.log('Word count:', wordCount);
```

3. Application guide

The application can be used in two ways:

- Using an API call tool
  The data can be retrieved and visualized within an API tool (for this application, **Insomnia** has been used. The user can input one of the two endpoints, http://localhost:3000/scrape or http://localhost:3000/count-words, depending on what is wanted to be done. The user also needs to start the NodeJS application, by typing in the command line of the IDE (for this project, **Visual Studio Code** has been used), **npm start.** In the Fetch Data API call (a PUT call), this line of code can be added in the JSON file to fetch the scraped data: "url": https://wsa-test.vercel.app/. In the Count Words API call, the user can retrieve the words from a blog post, by adding one of those blog post URLs:

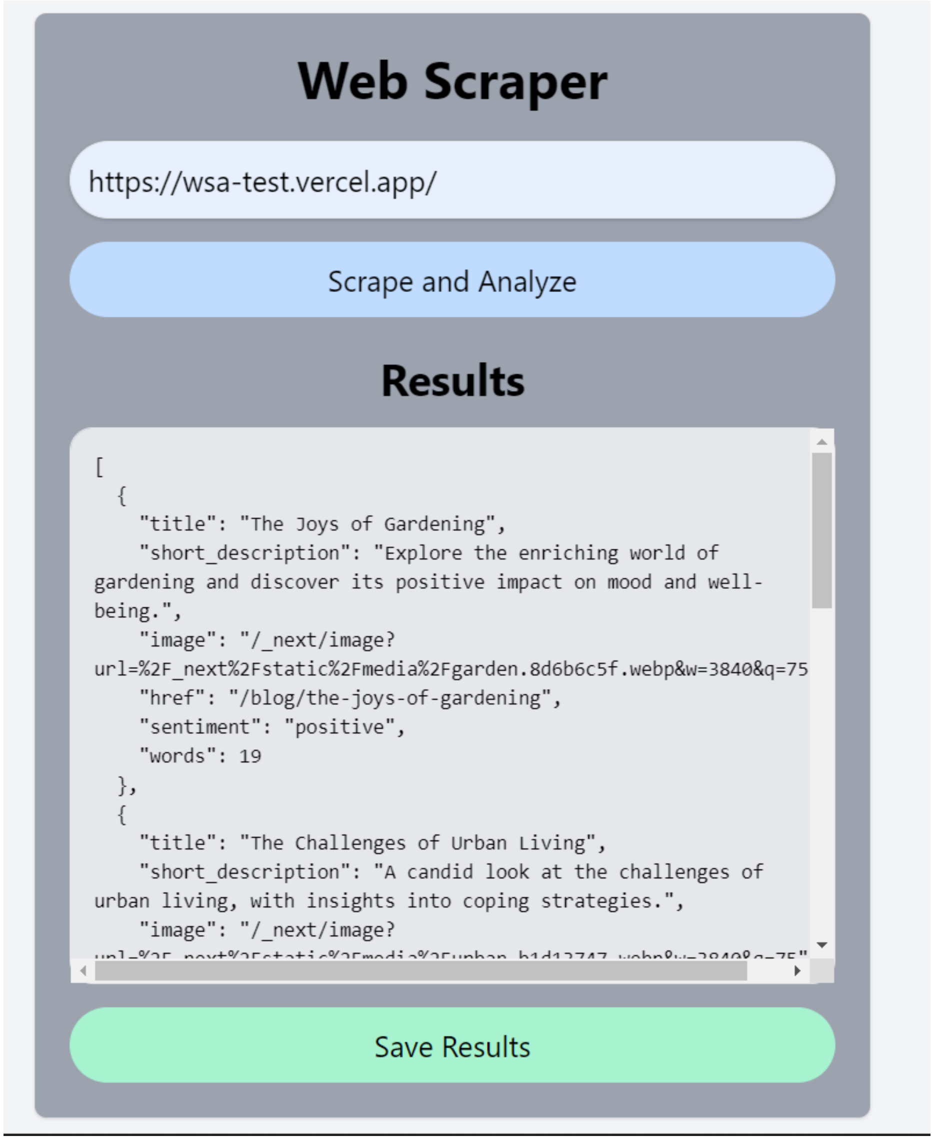
"url": "https://wsa-test.vercel.app/blog/the-joys-of-gardening"
"url": "https://wsa-test.vercel.app/blog/the-challenges-of-urban-living"
"url": "https://wsa-test.vercel.app/blog/the-radiant-days-of-summer"
"url": "https://wsa-test.vercel.app/blog/neutral-observations-on-modern-art"
"url": "https://wsa-test.vercel.app/blog/the-disappointing-reality-of-junk-food"



```
{
  "url": "https://wsa-test.vercel.app/"
}
```

```
[
  {
    "title": "The Joys of Gardening",
    "short_description": "Explore the enriching world of gardening and discover its positive impact on mood and well-being.",
    "image": "/_next/image?url=%2F_next%2Fstatic%2Fmedia%2Fgarden.8d6b6c5f.webp&w=3840&q=75",
    "href": "/blog/the-joys-of-gardening",
    "sentiment": "positive",
    "words": 19
  },
  {
    "title": "The Challenges of Urban Living",
    "short_description": "A candid look at the challenges of urban living, with insights into coping strategies.",
    "image": "/_next/image?url=%2F_next%2Fstatic%2Fmedia%2Furban.b1d13747.webp&w=3840&q=75",
    "href": "/blog/the-challenges-of-urban-living",
    "sentiment": "neutral",
    "words": 19
  },
  {
    "title": "The Radiant Days of Summer",
    "short_description": "Dive into the vibrant and joyful moments that the summer season brings to our lives.",
    "image": "/_next/image?url=%2F_next%2Fstatic%2Fmedia%2Fsummer.8f9285fa.webp&w=3840&q=75",
    "href": "/blog/the-radiant-days-of-summer",
    "sentiment": "positive",
    "words": 20
  },
  {
    "title": "Neutral Observations on Modern Art",
    "short_description": "An unbiased exploration of modern art, inviting readers to form their own opinions.",
    "image": "/_next/image?url=%2F_next%2Fstatic%2Fmedia%2Fart.038e828f.webp&w=3840&q=75",
    "href": "/blog/neutral-observations-on-modern-art",
    "sentiment": "neutral",
```

- Using the frontend application
  The provided frontend application can also be used for fetching and visualizing the required data. The user needs to start the NodeJS application, by typing in the command line of the IDE (for this project, **Visual Studio Code** has been used), **npm start.** Afterwards, it is needed to open a web browser page, and access **https://localhost:3000**. After the page has been loaded, it is required to input the URL of the target website (https://wsa-test.vercel.app/), and then press the “Scrape and Analyze” button. After a couple of seconds, the fetched data can be visualized in the output box.
  A functionality has been added, allowing the user to save the fetched data locally, under a .txt file. The web page can be seen below:



https://wsa-test.vercel.app/

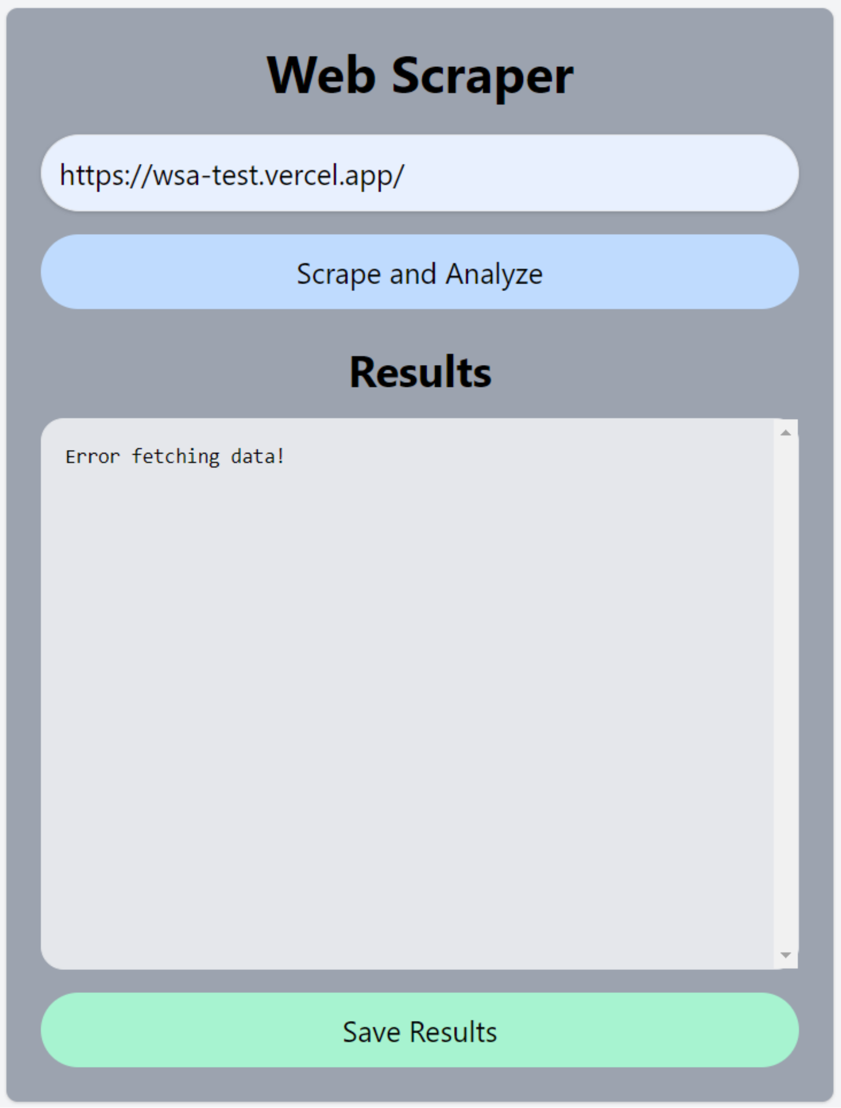
```
[
  {
    "title": "The Joys of Gardening",
    "short_description": "Explore the enriching world of
gardening and discover its positive impact on mood and well-
being.",
    "image": "/_next/image?
url=%2F_next%2Fstatic%2Fmedia%2Fgarden.8d6b6c5f.webp&w=3840&q=75
    "href": "/blog/the-joys-of-gardening",
    "sentiment": "positive",
    "words": 19
  },
  {
    "title": "The Challenges of Urban Living",
    "short_description": "A candid look at the challenges of
urban living, with insights into coping strategies.",
    "image": "/_next/image?
url=%2F_next%2Fstatic%2Fmedia%2Furban.b1d13747.webp&w=3840&q=75"
```

4. Known bugs/issues

The application is returning the required data in most of the cases, but a bug has been observed.

At times, after starting the application and accessing the URL with the port 3000 and after pressing the Scrape and Analyze button, a non-expected behaviour occurs. The application is throwing an error “Error fetching data!”. This issue is most probably caused by the dynamic loading of the page, even though awaits have been added, to allow the page to load all of its components. A solution was tried, to allow the application to **retry multiple times** to send the request, but with no success, so the solution was dropped. A workaround to fix this issue can be to reload the frontend page and try again by pressing the Scrape and Analyze button, or pressing the Scrape and Analyze button multiple times.

Even though this issue appears at times, the result is the expected one.



https://wsa-test.vercel.app/

```
Error fetching data!
```