**CHAPTER 1 - INTRODUCTION**

**1.1 Introduction of Web Scraping with Python and Flask**

In today's data-driven world, information is abundant on the internet, and extracting valuable insights from the web can be crucial for various applications. Web scraping is a powerful technique that allows us to automate the extraction of data from websites. In this project, we aim to develop a web scraping application using Python and Flask, a web framework for building web applications.

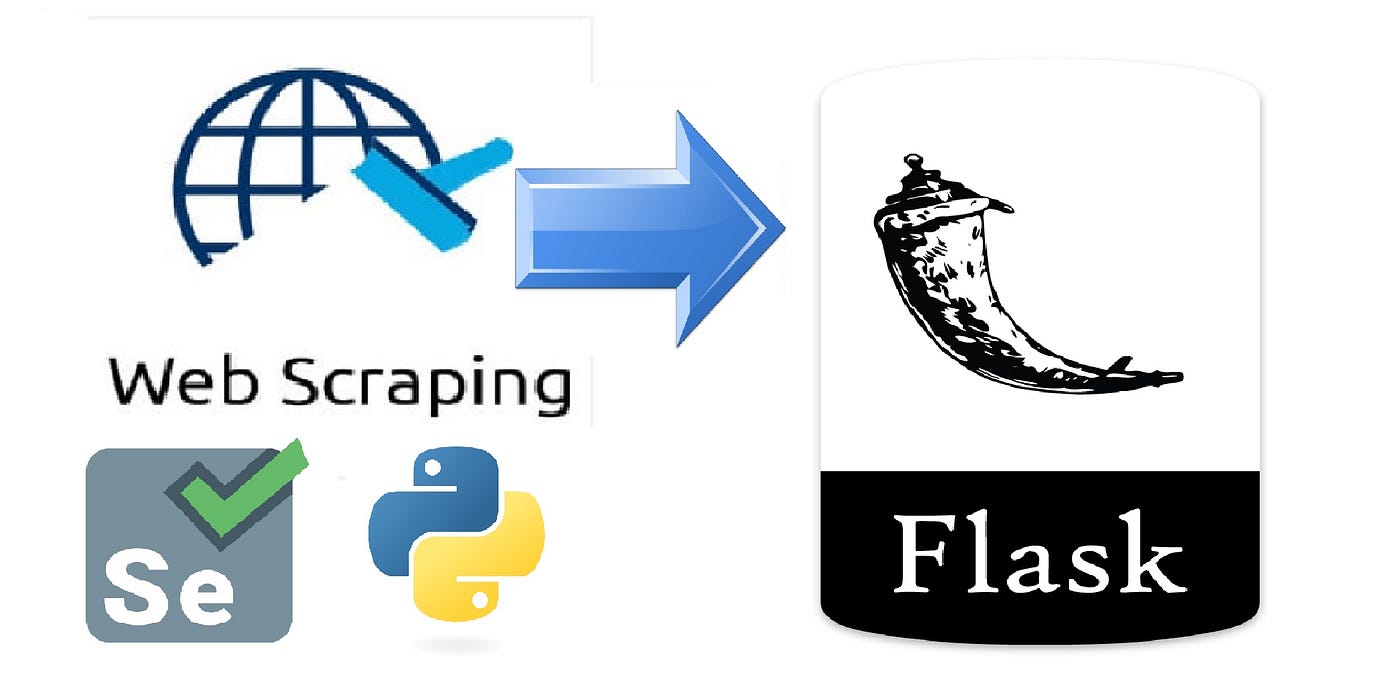


Fig. 1.2 Web Scraping

**1.2 Objective**

1. Data Extraction: The primary goal of the project is to create a web scraper that can navigate through websites, extract relevant data, and store it for further analysis. This could include scraping product details, news articles, or any other publicly available information.

2. User Interface with Flask: Implement a user-friendly web interface using Flask that allows users to input URLs, specify data extraction parameters, and initiate the scraping process. Flask will serve as the backend framework to handle user requests and manage the scraping tasks.

3. Data Storage: Develop a mechanism to store the scraped data efficiently. This could involve using a database such as SQLite or MongoDB to organize and manage the collected information.

4. Scalability: Design the web scraper to be modular and scalable, enabling users to easily add new websites for scraping by extending the functionality of the application.

**1.3 Technologies Used:**

1. Python: The core programming language for building the web scraper logic and backend functionality.

2. Flask: A lightweight web framework for Python, used to create the user interface and handle HTTP requests.

3. Beautiful Soup: A Python library for pulling data out of HTML and XML files. It will be used for parsing and navigating the HTML content of web pages.

4. Requests: A Python library for making HTTP requests. It will be used to fetch the HTML content of web pages.

5. Database (e.g., SQLite or MongoDB): To store and manage the scraped data.

**BEAUTIFUL SOAP**

Beautiful Soup is a library that makes it easy to scrape information from web pages. It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.

pip install beautifulsoup4 install beautifulsoup4

>>> from bs4 import BeautifulSoup

>>> soup = BeautifulSoup("<p>Some<b>bad<i>HTML")

>>> print(soup.prettify())

<html>

<body>

<p>

Some

<b>

bad

<i>

HTML

</i>

</b>

</p>

</body>

</html>

>>> soup.find(text="bad")

'bad'

>>> soup.i

<i>HTML</i>

#

>>> soup = BeautifulSoup("<tag1>Some<tag2/>bad<tag3>XML", "xml")

#

>>> print(soup.prettify())

<?xml version="1.0" encoding="utf-8"?>

<tag1>

Some

<tag2/>

bad

<tag3>

XML

</tag3>

</tag1>

**REQUEST**



In this guide for [*The Python Web Scraping Playbook*](https://scrapeops.io/python-web-scraping-playbook/)*,* we will look at how to set up your python requests scrapers to avoid getting blocked, retrying failed requests and scaling up with concurrency.

Python Requests is the most popular HTTP client library used by Python developers, so in this article we will run through all the best practices you need to know. Including:

* [Making GET Requests](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#making-get-requests)
* [Making POST Requests](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#making-post-requests)
* [Using Fake User Agents With Python Requests](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#using-fake-user-agents-with-python-requests)
* [Using Proxies With Python Requests](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#using-proxies-with-python-requests)
* [Retrying Failed Requests](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#retrying-failed-requests-with-python-requests)
* [Scaling Your Scrapers Using Concurrent Threads](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#scaling-your-python-request-scrapers-with-concurrent-threads)
* [Rendering JS On Client-Side Rendered Pages](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#rendering-js-on-client-side-rendered-pages)

## Making GET Requests[​](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#making-get-requests)

Making GET requests with python requests is very simple.

We just need to request the URL using requests.get(url):

The following are the most commonly used attributes of the Response class:

1. status\_code: The HTTP status code of the response.
2. text: The response content as a Unicode string.
3. content: The response content in bytes.
4. headers: A dictionary-like object containing the response headers.
5. url: The URL of the response.
6. encoding: The encoding of the response content.
7. cookies: A RequestsCookieJar object containing the cookies sent by the server.
8. history: A list of previous responses if there were redirects.
9. ok: A boolean indicating whether the response was successful (status code between 200 and 399).
10. reason: The reason phrase returned by the server (e.g., "OK", "Not Found").
11. elapsed: The time elapsed between sending the request and receiving the response.
12. request: The PreparedRequest object that was sent to the server.

## Making POST Requests[​](https://scrapeops.io/python-web-scraping-playbook/python-requests-web-scraping-guide/#making-post-requests)

Making POST requests with python requests is also very simple.

To send **JSON data** in a POST request, we just need to request the URL using requests.post() along with the URL and the data using the json parameter:

import requests  
  
url = 'http://quotes.toscrape.com/'  
data = {'key': 'value'}  
  
*# Send POST request with JSON data using the json parameter*  
response = requests.post(url, json=data)  
  
*# Print the response*  
print(response.json())

**1.4 Workflow**

1. User Input: Users input the URL of the website they want to scrape and specify the data they are interested in.

2. Scraping Logic: The web scraper, powered by Beautiful Soup, navigates through the HTML structure of the provided URL, extracts the specified data, and processes it.

3. Data Storage: The scraped data is stored in a database for easy retrieval and analysis.

4. User Interface: Flask provides a user interface where users can interact with the application, monitor the scraping process, and access the collected data.

**1.5 Conclusion**

This project aims to showcase the power of web scraping using Python and Flask, providing users with a versatile tool to extract valuable information from the web efficiently. The combination of a robust scraping engine, a user-friendly interface, and data storage capabilities makes this project a valuable asset for researchers, analysts, and anyone seeking to harness the wealth of data available on the internet.

1. **CHAPTER - LITERATURE SURVEY**

Web scraping and web development using Python and Flask have gained significant attention in both academia and industry. Various research papers, articles, and tutorials explore different aspects of web scraping, data extraction, and web application development. Here is a brief literature survey covering key points relevant to this project:

1. Web Scraping Techniques:

- "Web Data Extraction Techniques: A Systematic Literature Review" by Rizwana Kausar et al. provides an overview of different web scraping techniques and methodologies.

- "A Survey of Web Information Extraction Systems" by R. Khusro and A. Kumar explores various methods used for extracting information from the web.

2. Python for Web Scraping:

- "Python Web Scraping: A Comprehensive Guide" by Ryan Mitchell is a widely recognized book that delves into the basics of web scraping using Python.

- "Web Scraping with Python: A Comprehensive Guide" by J.R. Parker provides practical insights into Python-based web scraping and data extraction.

3. Flask for Web Development:

- "Flask Web Development: Developing Web Applications with Python" by Miguel Grinberg is a comprehensive guide on web development using Flask.

- "Building Web Applications with Flask" by Italo Maia offers practical examples and best practices for developing web applications with Flask.

4. Best Practices and Ethics in Web Scraping:

- "Web Scraping: A Review of Best Practices" by Frank McCown et al. discusses ethical considerations, challenges, and best practices in web scraping.

- "Legal and Ethical Issues of Web Scraping" by Sebastian Manhart explores the legal and ethical aspects associated with web scraping activities.

5. Data Storage and Management:

- "Data Storage Techniques in Web Scraping" by A. Jain and S. Kumar surveys different methods for storing and managing data obtained through web scraping.

- "Web Scraping and Databases: A Survey" by V. Lakshmi and N. Meenakshi provides insights into integrating web scraping with various database systems.

6. Scalability in Web Scraping:

- "Scalable Web Scraping with MongoDB and Python" by M. Khan provides guidance on building scalable web scraping solutions using MongoDB as a backend storage system.

- "Scalable Web Scraping Using Distributed Systems" by S. Ravi et al. explores the use of distributed systems to enhance the scalability of web scraping applications.

These references cover a range of topics related to web scraping, Python programming, Flask development, ethical considerations, and data storage techniques. By leveraging insights from these works, this project aims to integrate best practices and contribute to the existing body of knowledge in the field of web scraping and web application development.

1. **CHAPTER - METHODOLOGY USED**

The methodology for developing the Web Scraping with Python and Flask project involves several key steps, from planning and design to implementation and deployment. Here's an overview of the methodology used in this project:

1. \*\*Requirements Analysis:\*\*

- Identify the project's goals and objectives.

- Define the scope of the web scraping application, specifying the types of websites it will support and the data to be extracted.

2. \*\*Technology Stack Selection:\*\*

- Choose Python as the core programming language for its versatility and extensive libraries.

- Select Flask as the web framework to build the user interface and handle backend functionalities.

- Opt for Beautiful Soup and Requests libraries for web scraping operations.

3. \*\*System Design:\*\*

- Define the architecture of the web scraping application, outlining the components and their interactions.

- Design the user interface with Flask templates to ensure a responsive and user-friendly experience.

- Plan the database schema for storing scraped data efficiently.

4. \*\*Web Scraping Logic:\*\*

- Implement web scraping logic using Beautiful Soup to parse HTML content.

- Utilize the Requests library to fetch HTML content from specified URLs.

- Handle dynamic content loading and pagination if necessary.

- Extract relevant data based on user-defined parameters.

5. \*\*User Interface Development:\*\*

- Use Flask to create routes and views for the web application.

- Design HTML templates for user input forms, result displays, and progress tracking.

- Implement interactivity with JavaScript if required for a smoother user experience.

6. \*\*Database Integration:\*\*

- Integrate a database (e.g., SQLite or MongoDB) to store the scraped data.

- Develop functions to interact with the database, including storing and retrieving data.

7. \*\*Scalability Considerations:\*\*

- Design the web scraper to be modular, allowing easy addition of new scraping modules for different websites.

- Consider scalability by optimizing code and adopting best practices for handling large volumes of data.

**4.CHAPTER – VIRTUAL ENVIRONMENT**

**Virtual environments**

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system’s packages.

### **Create an environment**

Create a project folder and a venv folder within:

**For Window**

> mkdir myproject

> cd myproject

> py -3 -m venv venv

**For MacOS linux**

$ mkdir myproject

$ cd myproject

$ python3 -m venv venv

**INSTALL FLASK**

Within the activated environment, use the following command to install Flask:

$ pip install Flask

The flask command is a CLI for interacting with Flask apps. The [docs](http://flask.palletsprojects.com/cli/) describe how to use CLI commands and add custom commands. The flask run command is the preferred way to start the development server.

Never use this command to deploy publicly, use a production WSGI server such as Gunicorn, uWSGI, Waitress, or mod\_wsgi.

As of Flask 2.2, use the --app option to point the command at your app. It can point to an import name or file name. It will automatically detect an app instance or an app factory called create\_app. Use the --debug option to run in debug mode with the debugger and reloader.

Prior to Flask 2.2, the FLASK\_APP and FLASK\_ENV=development environment variables were used instead. FLASK\_APP and FLASK\_DEBUG=1 can still be used in place of the CLI options above.

$ export FLASK\_APP=sample

$ export FLASK\_ENV=development

$ flask run

On Windows CMD, use set instead of export.

> set FLASK\_APP=sample

For PowerShell, use $env:.

> $env:FLASK\_APP = "sample"

The python sample.py command runs a Python file and sets \_\_name\_\_ == "\_\_main\_\_". If the main block calls app.run(), it will run the development server. If you use an app factory, you could also instantiate an app instance at this point.

if \_\_name\_\_ == "\_\_main\_\_":

app = create\_app()

app.run(debug=True)

**5.CHAPTER - Source code of project**

1. **HTML Code**

<!DOCTYPE html>

<html lang="en" style="background-image: url('../static/image/wallpaperflare.com\_wallpaper.jpg');

background-attachment: fixed;">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>TecNews Post</title>

<!--<link rel="stylesheet" href="{{url\_for('static',filename='css//style.css')}}">-->

<link href="https://unpkg.com/tailwindcss@^2/dist/tailwind.min.css" rel="stylesheet">

<body>

<div class="mb-10 top-1 w-screen flex flex-col justify-center items-center">

<img src="../static/image/logo.png" class="w-64 h-64" alt="Logo Image">

</div>

<div class="bottom-6 m-auto relative" id="container" style="width:614px; height:700px;">

<img src="" id="bg1" width="614px" height="700px">

<span class=" font-mono text-lg absolute text-white whitespace-pre-line text-justify" id="News"

style="width: 550px; top: 100px; left: 30px; line-height: 2em;">

{{News|safe}}

<!-- These curly braces are a gateway for python variables and functions (via flask) to Front-end. This is done with the help of Jinja template engine.-->

</span>

</div>

<div id="canvasWrapper" class="mb-10 w-screen flex justify-center items-center">

<button

class="text-white p-3 rounded-md font-sans text-xl bg-gradient-to-r from-green-400 to-blue-500 hover:from-pink-500 hover:to-yellow-500"

onclick="download()">Download Post</button>

</div>

<script>

var newbg = document.getElementById("bg1");

var background = new Image();

background.src = "static/image/" + (Math.floor(Math.random() \* (7)) + 1) + ".jpg";

newbg.src = background.src;

console.log(newbg.src);

</script>

<script src="/static/js/h2c.min.js"></script>

<script>

function download() {

html2canvas(document.getElementById("container"), { height: 700, width: 614 }).then(canvas => {

//document.body.appendChild(canvas);

var a = document.createElement('a');

a.href = canvas.toDataURL("image/jpeg").replace("image/jpeg", "image/octet-stream");

a.download = 'somefilename.jpg';

a.click();

});

}

</script>

</body>

</html>

1. **CSS Code**

.gfg {

margin: 3%;

position: relative;

height: 700px;

width: 614px;

}

.first-txt {

font-family: monospace;

font-size: large;

position: absolute;

line-height: 2em;

top: 100px;

left: 30px;

color: rgb(255, 255, 255);

width: 550px;

white-space: pre-line;

text-align: justify;

}

1. **JavaScript Code**

!function(A,e){"object"==typeof exports&&"undefined"!=typeof module?module.exports=e():"function"==typeof define&&define.amd?define(e):(A="undefined"!=typeof globalThis?globalThis:A||self).html2canvas=e()}(this,function(){"use strict";

1. **Python Code**

from flask import Flask,url\_for,render\_template,request

from bs4 import BeautifulSoup

import requests

app = Flask(\_\_name\_\_)

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

def index():

url = "https://www.businesstoday.in/technology/news"

req = requests.get(url)

soup = BeautifulSoup(req.content, 'html.parser')

#outerData = soup.find\_all("div",class\_="widget-listing",limit=6)

#print(outerData)

finalNews=""

for data in soup.find\_all("div",class\_="widget-listing",limit=6):

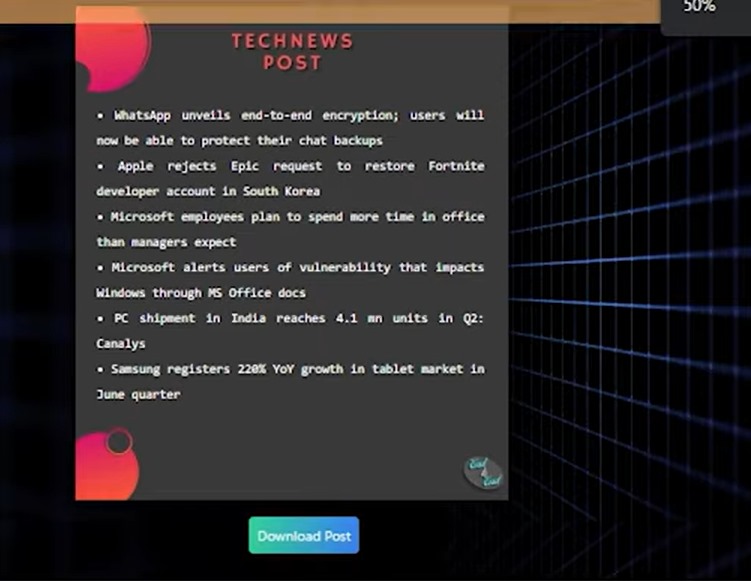
news=data.div.div.a["title"]

finalNews += '\u2022 '+news+'\n'

#print(finalNews)

return render\_template("index.html",News=finalNews)

**6.CHAPTER – Conclusion and Recommendation**

****

**7. CHAPTER - Conclusion and Recommendation**

The"Web Scraping with Python and Flask" project has successfully developed a versatile and user-friendly tool for automating data extraction from the web. Leveraging Python's rich libraries and Flask's web framework, the application offers an intuitive interface for users to specify data extraction parameters and seamlessly navigate through websites. The modular and scalable design, coupled with efficient data storage capabilities, ensures adaptability to diverse requirements and ease of managing collected information. The comprehensive documentation facilitates easy installation and usage, while ethical considerations underscore responsible scraping practices. Moving forward, it is recommended to continue refining the application, addressing potential security concerns, and exploring further optimization for enhanced performance. Additionally, the project could benefit from community contributions and continuous updates to accommodate evolving web technologies and user needs.

**References**

• https://www.w3schools.com/

• https://careerfoundry.com/en/tutorials/web-development-for-beginners/introduction-to-web-development/

• https://www.Udemy.com/bootstrap/

• https://www.codecademy.com/catalog/subject/web-development

• https://www.geeksforgeeks.org/web-development/

• https://getbootstrap.com/

• https://www.javatpoint.com/html-tutorial

• https://www.javatpoint.com/css-tutorial

• https://www.javatpoint.com/javascript-tutorial

• https://www.javatpoint.com/bootstrap-tutorial

• https://stackoverflow.com/