UNIVERSITY POLITEHNICA OF BUCHAREST

Faculty of Electronics, Telecommunications and Information Technology

Project 3

Web Scraper

|  |  |
| --- | --- |
| Coordinating professor:  Anamaria Dumitrescu | Students:  Dionisie Crina-Ioana 442G  Mircia Flaviu-Constantin 442G |

**Table of Contents**

[**Introduction** 1](#_Toc152789133)

[**Chapter 1. Tools and Frameworks** 2](#_Toc152789134)

[**1.1 Python** 2](#_Toc152789135)

[**1.2 Beautiful Soup and ‘requests’ libraries** 2](#_Toc152789136)

[**1.3. Django Framework** 3](#_Toc152789137)

[**1.4. GitHub** 4](#_Toc152789138)

[**Chapter 2. Project Workflow** 5](#_Toc152789139)

[**Chapter 3. Conclusions** 10](#_Toc152789140)

[**Bibliography** 11](#_Toc152789141)

**List of Figures**

[Fig. 1 Basic overview of the search-engine architecture 5](#_Toc152787353)

[Fig. 2 Overview of the application work-flow. 6](#_Toc152787354)

[Fig. 3. The project input form 9](#_Toc152787355)

[Fig. 4. The results of the searched word 9](#_Toc152787356)

# **Introduction**

In the era of e-commerce, web scraping has emerged as a critical technology for data acquisition and analysis. Our Python project represents an effective combination of web scraping techniques and the Django framework to enable efficient data extraction from a popular online platform for buying and selling a wide range of products, Lajumate.ro. In this documentation, we describe the functionality of our project, highlighting its significance and potential applications.

Access to real-time data from online marketplaces has become a key factor in informed decision-making. Lajumate.ro, a source of information for both consumers and researchers, contains a lot of information. This project serves as a bridge between users and the vast repository of listings on this site. The significance of web scraping lies in its ability to provide access to crucial product details, such as titles and prices, which can be instrumental in market research, price comparison, and locating the best deals.

Our project leverages advanced web scraping techniques to scrape Lajumate.ro efficiently. It utilizes Django, a high-level Python web framework, to transform this raw data into a structured and user-friendly format. The integration of Django significantly enhances the user experience by offering a seamless and organized interface for data retrieval and analysis. With these two powerful tools, we designed a robust and versatile tool that allows users to extract essential information such as product titles and prices.

The core functionality of our project centers on conducting searches on Lajumate.ro, allowing users to retrieve relevant product listings based on their specified search criteria. By inputting specific keywords, users can initiate searches, and the system will swiftly retrieve product details that match the search parameters. The key information extracted provides users with valuable insights into the offerings on Lajumate.ro. This documentation will guide users through the project, enabling them to explore and utilize the web scraping capabilities effectively.

Moreover, our project underscores the importance of collaboration and version control in academic and professional settings. To enhance the project's accessibility and maintainability, we have integrated it with GitHub, a widely used version control platform.

In summary, our Python project represents a pivotal resource for data extraction and analysis from the site Lajumate.ro. Web scraping and the Django framework harmoniously merge to facilitate this process, making it user-friendly and powerful.

# **Chapter 1. Tools and Frameworks**

## **1.1 Python**

Python is a dynamic, high-level programming language that combines readability, versatility, and simplicity to become immersed in the coding community. Its roots trace back to the late 1980s, and since then, it has grown rapidly into a language of choice for developers across diverse domains [1]. What sets Python apart is its emphasis on code readability and a straightforward syntax that mirrors the clarity of human thought. Guido van Rossum, Python's creator, envisioned a language that prioritizes simplicity, fostering a community where developers can focus on problem-solving rather than struggling with complex syntax [2].

One key factor for its success is its adaptability. It seamlessly embraces various programming paradigms, including procedural, object-oriented, and functional programming. This adaptability positions Python as a universal tool for projects ranging from web development to scientific computing and artificial intelligence [1]. The language's dynamic typing and automatic memory management contribute to its user-friendly nature, allowing developers to focus on the logic of their programs rather than dealing with low-level details.

Talking about the syntax, the code readability is another key factor. A clean, simple style is promoted by using indentation for block delimiters and using less punctuation [2]. This focus on readable code translates into ease of collaboration. In projects where multiple developers contribute, Python's syntax becomes a shared language, facilitating communication and making it an ideal choice for teamwork.

Python's strength is amplified by its extensive standard library and a vast ecosystem of third-party packages. The inclusion of modules for various tasks within the standard library reduces the need for developers to build functionality from scratch, promoting efficiency and code reuse [1]. Furthermore, the Python Package Index (PyPI) boasts a repository of packages covering a lot of domains, enabling developers to access pre-built solutions for diverse requirements.

Due to its active community of worldwide developers who contribute to an open, collaborative environment where knowledge sharing is essential, Python becomes a strongly appreciated programming language. Online forums, documentation, and an abundance of tutorials provide a wealth of resources for both begginer and experienced developers [1]. This collaborative spirit ensures that Python remains a living language, evolving with the needs and innovations of its global user base.

## **1.2 Beautiful Soup and ‘requests’ libraries**

In the realm of web scraping, Python emerges as a versatile and powerful language, providing developers with a suite of tools to navigate the complexities of data extraction. The ‘requests’ library and Beautiful Soup, each adding their own distinct contribution to the coordination of web scraping jobs, are the foundation of this capability [3]. The clarity and simplicity that Python's founder, Guido van Rossum, promoted these libraries' functionality and design [2].

The ‘requests’ library provides a straightforward mechanism for making HTTP requests [4]. It supports various HTTP methods, including GET and POST, providing flexibility for different scraping scenarios. Additionally, it seamlessly manages cookies, sessions, and headers, further streamlining the interaction with web servers. The language's general concept that code should be understandable and accessible is reflected in its intuitive syntactic design. This simplicity enables developers to focus on the core of their scraping tasks, emphasizing functionality without compromising readability.

The Beautiful Soup library enhances ‘requests’'s ability to handle requests. When it comes to processing HTML and XML documents, Beautiful Soup is remarkable at transforming raw markup into a navigable tree structure. Adaptability, an essential quality in the ever-changing internet ecosystem, is ensured by its capacity to handle HTML that is not properly written [5]. The association between ‘requests’ and Beautiful Soup is a perfect example of how Pythonic programming combines clarity and usefulness.

In practice, the workflow often involves using 'requests' to retrieve HTML content from a website, followed by Beautiful Soup's application to parse and navigate the obtained markup. Developers can then extract relevant information based on tag structures, attributes, or other patterns within the parse tree. This collaborative usage exemplifies the synergy between the two libraries, demonstrating how Python's ecosystem provides cohesive solutions for complex tasks such as web scraping.

## **1.3 Django Framework**

Django, a high-level web framework for Python, stands as a robust and versatile tool for building web applications. Its origins trace back to a development effort at the Lawrence Journal-World newspaper in 2003, where Django's creators sought a framework that prioritized rapid development, maintainability, and the implementation of the DRY (Don't Repeat Yourself) principle [6]. Django encapsulates the Model-View-Controller (MVC) architectural pattern, emphasizing modularity and separation of concerns. It equips developers with an extensive set of components, including an Object-Relational Mapping (ORM) system, templating engine, and an administration panel which facilitates efficient and streamlined web development [7].

The framework's ORM system simplifies database interactions by mapping Python objects to database tables, abstracting away the complexities of SQL queries. This abstraction enhances code readability and accelerates the development process, aligning with Django's commitment to code simplicity and clarity [6]. The templating engine empowers developers to create dynamic web content seamlessly, encouraging the separation of presentation and logic. Moreover, Django's security features, such as protection against common web vulnerabilities, contribute to the framework's reputation as a reliable choice for building secure web applications [6].

In the context of web scraping, Django's capabilities extend beyond traditional web application development. While it is not primarily designed for web scraping, its robust architecture provides a foundation for integrating web scraping functionalities into Django projects. Developers can leverage Django's ORM and views to manage scraped data efficiently and present it through the framework's templating system. The extensibility of Django allows developers to create custom management commands or dedicated Django apps for handling specific web scraping tasks [8, 9].

Web scraping and Django's competence in web development create an effective combination. Using Django, developers may create comprehensive solutions that meet a wide range of data-driven needs by combining web scraping functionality with dynamic web applications.

## **1.4 GitHub**

Version control systems are essential for providing structure, traceability, and collaborative efficiency in the complex world of software development. Git is one of these tools that has become indispensable, revolutionizing the way developers approach code management. Version control, at its heart, helps developers to track changes, work together smoothly, and maintain project integrity by addressing the difficulties involved in managing changes in software projects.

Git, a distributed version control system, has revolutionized the way developers manage and collaborate on projects. Initially developed by Linus Torvalds, the creator of Linux, Git allows developers to track changes in source code during software development [9, 10]. What sets Git apart is its decentralized nature, enabling multiple developers to work on a project simultaneously. Each developer maintains a local copy of the entire project history, making it efficient and resilient.

GitHub, a web-based platform built around Git, amplifies the collaborative potential of version control. Serving as a central repository for projects, GitHub facilitates seamless collaboration among developers worldwide [10].

The collaborative nature of GitHub extends beyond version control. It provides tools for issue tracking, project management, and code review, offering a comprehensive ecosystem for software development. The development workflow is further improved by GitHub's association with continuous integration technologies, which guarantee that modifications made by various developers are compatible and preserve project stability [10]. This collaboration makes Git and GitHub indispensable tools in the modern software development landscape.

# **Chapter 2. Project Workflow**

The project architecture is based to a similar architecture to the microservice one, but with a twist. Because it is a relatively small project, in terms of number of lines of code, we choose not to take a multiple-repositories architecture, putting everything in a single repository.  
 Given the above choice, we decided not to include a message queue for communication between microservices because it will add seriously overhead in terms of code complexity.

An overview of to chosen architecture can be seen in the following schematic:

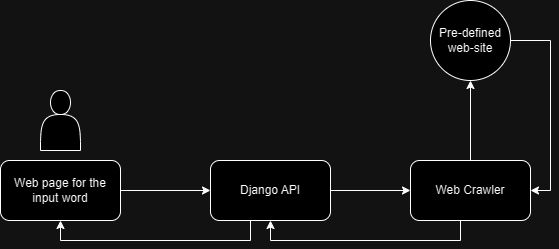
**

Fig. 1 Basic overview of the search-engine architecture

We put emphasis on the “Pre-defined” because how most of the modern web-sites are build. The problem encountered with bigger and modern sites (e.g: eMag), in terms of web-scraping, is the captcha verification which needs to be bypassed in order for the web-scraping tool to work. This human verification test put a stop to our intiative to scrape this (and many more) websites. So, as a work-around we have tested and choose “lajumate.ro”.

In the following pages, we will define and split this architecture between two major components:

* The Django REST api
* The Web-crawler

This split is necessary for a better understanding of the chosen architecture because it can be easily migrated to a microservices architecture if the project is futurely developed.

The Django REST API is the communication and validation pipeline between the user and the web-crawler. The user writes a word (or more words) in the home page of the webserver, there is a simple Javascript program that takes the word from the input form, overrides the default request of the form (the default being a POST request, we need a GET), then serializes the word into a JSON format and sends the HTTP request to the given endpoint. When it reaches the Django API, we make validation and sanitization of the given word and then we pass the payload to the web-crawling application. When the webscraping is done, the results are returned and again serialized from a String format to a JSON one, after the serialization is done the application renders the webpage and redirects the user to it.

To be able to make the connection between the two HTML pages, the user will interact with, the application should be able to localize the relative path and make them available to the endpoint that is accessible for the user. Due to this fact, we choose to render the pages on the go, making them templates instead of simple, static web-pages.

For a better understanding of the above statements, the following figure was added with description of the workflow.

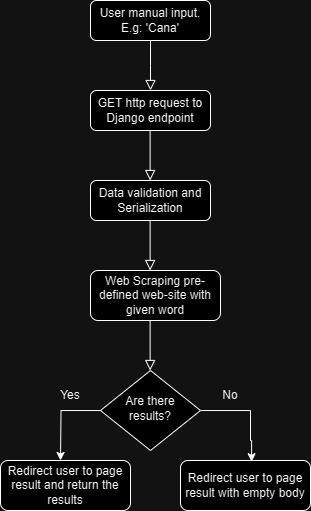


Fig. 2 Overview of the application work-flow.

To understand this workflow, we need to get a few paragraphs back where the terms “GET” and “POST” appeared. These two words are commonly known as “HTTP verbs” or “HTTP methods” and they are responsable for basic communication between to endpoints (in our case the two endpoints are the user and the actual endpoint of the Django api).

A GET request represents a way of requesting resources from that specific web-server. The resources varies from simple HTML pages, that are also retrieved with a request from the server, to the very complex JSON Web Token authetication (which is out of scope for the current project, but very in handy to know).

The POST request is a way of sending resources to the webserver, for example an order for an online-shop. We have decided to replace the default POST verb of the HTML button component with a GET verb, with the help of JavaScript, because of the easier handling in the Django REST API.

We have also choose of not putting query parameters in the GET request to the API and mapping the URL to “host-name:port/api\_url/<word>” to be easier for parsing the request in the Controller-like component of Django. We define the Controller-like component like that because Django is not a Model-View-Controller framework and more of Model-View-Template, which is pretty similar to the MVC architectural pattern, but instead of the Controller we have the corresponding View and instead of the classical View in the MVC, we have the Template, which we described above as dinamically generated HTML page.

The HTTP methods/verbs follow a complex logic that have literally books written to explain them. Those explanation are, again, out-of-scope for the purpose of this project, but we need to understand, in simple terms, how those methods works.

When we type a URL in a browser, there are many things that happens. First of all, we need to understand of that URL is build.

The web URL (Uniform Resource Locator) is a string of characters used to address and locate resources on the internet. It's essentially a web address that specifies the location of a resource, such as a web page, image, document, or any other file on the internet and it contains various components:

* Scheme: This indicates the protocol used to access the resource. Common schemes include HTTP, HTTPS, FTP, etc.
* Domain or Host: This part of the URL specifies the location of the resource. It can be an IP address or a domain name (e.g., www.example.com).
* Port (optional): If the resource is accessed via a specific port on the server, it's mentioned in the URL. For HTTP, the default port is 80, and for HTTPS, it's 443. If these ports are used, they are often omitted from the URL.
* Path: The path identifies the specific resource or file on the server. It comes after the domain and typically represents the structure of the website's directories (e.g., /blog/article1).
* Query Parameters (optional): These parameters, separated by a question mark ?, provide additional information to the server about the request. They consist of key-value pairs (e.g., ?page=1&sort=asc).
* Fragment (optional): The fragment is indicated by a hash # and points to a specific section within a resource (e.g., #section2).

The URL is then passed to the local DNS which is located, most of the times, in the Operating System (e.g: Unix, Linux, Windows) or in the Router. The Domain Name System is responsable for mapping the URL to an actual Ip Address for it to be routed and passed around until it finally reaches the destination.

With the URL, there exists most of time a HTTP method. The HTTP method, has as destination the URL, multiple headers and a body.

The headers are responsable for negotiating the type of body format, for example, extremly important for the serialization process.

The body or the payload is the actual information that is going to be serialized by the each endpoint (browser or webserver).

We kept repeating the word “Serialization” which is just a fancy word for taking a payload with a format and transforming it into another format. For example, we serialize a word when we write a String type word and we need a JSON format. The reverse operation is called “Deserialization”.

Returning to our application, we need to describe the core of the application which is the web-scraper.

The web-scraper or web-crawler is resposanble for going to a specific webpage and taking the source structure of that page (commonly being represented by HTML).

The web-crawler is an efficient tool for parsing the Document Object Model (DOM) into simple strings and extracting the value of it. It is based on a simple, but rather long to implement, tree traversal of the DOM, where each data type (e.g: <body>, <h1>, <p>) might be a child to a superior data type. For parsing the DOM with the above method, there are, mainly, two approaches. One is a standard algorithm called “Depth-first search” which basically, as the name says, traverse the tree first in-depth. The algorithm works as such: from the root node, which in our case will be the <html> tag, it recursively goes to each of the children nodes until it reaches the leaves nodes (that are defined as nodes with no children). Depending on what variations of the DFS algorithm it is implemented the extraction of the value from the html tag could be done when the respective node is reached or after the recursion stack is done (pre-order traversal vs post-order traversal).

The other algorithm for traversing the tree is the “Breadth-first search” which focuses on discovering the elements on each level then traversing the next one. The usual algorithm is implemented iterative with the help of a First-in First-out Queue where each parent node has one or multiple children nodes and those children nodes have neighbor nodes (which are the children of the same parent node except the current one). To successfully implement this algorithm it also necessary to have a set/list of visited nodes, otherwise it will possibly run into an inifinte loop.

Another approach, for extracting these values is the Regular Expressions, however it is NOT recommended because of how the HTML is constructed. With the HTML format (similar to the XML) having basically unlimited keywords, with the only rule being to be contained between “< >”, given this fact parsing the DOM with regular expression can be dissatrous in terms of returned data.

In our application, we have choose to use a pre-built web-scraper, due to the fact that building a DOM parser with a Tree Traversal is a tidious task, named BeautifulSoup or bs4. The BeautifulSoup library is a simple way of traversing and extracting the necessary data of the HTML structure. Concretely, we parsed with BeautifulSoup the web page resulted from the GET request of the website “lajumate.ro” with the query parameter that our original user put in the search form.

In addition to these details, we have also implemented a minimalist front-end for our search and results forms. The front-end is build with the help of Boostrap library for CSS and plain CSS for a better customization. Boostrap is a free and open-source CSS framework which is very useful for creating a modern UI. In our case, we have created flex-boxes on top of the divs which allow us to manipulate with ease the “boxes” in our front-end pages. The CSS files are statically stored inside the Django project and then loaded with the dynamically generated templates.

In the final form the project looks like this:

A screenshot of a computer

Description automatically generated

Fig. 3. The project input form

**A screenshot of a computer

Description automatically generated**

Fig. 4. The results of the searched word

# **Chapter 3. Conclusions**

As we bring to an end our study of web scraping and data extraction, our project provides proof of the combination of technical knowledge and problem-solving skills. Using Python's Beautiful Soup and Requests modules in-depth, we concentrated on browsing and retrieving specific data from the dynamic Lajumate.ro site. This activity, complex and challenging, required an equal amount of technical proficiency and an effective understanding of web architecture.

This project's integration of frontend accessibility and backend functionality was one of its most interesting feature. We were able to develop a simplified user interface that made it easier for users to engage with the scraping tool by integrating Django, a reliable web framework. This combination closed the gap between complex data extraction methods and end-user convenience by turning a strictly technical solution into an approachable tool.

Beyond the technical complexities, this project provided valuable insights into data filtration and refinement. The ability to retrieve particular data from an extensive dataset demonstrated the effectiveness of online scraping methods and the need of focused data collection. Decision-making processes in a variety of fields, including market analysis and product development, can be greatly improved by this ability to sort through enormous volumes of data and extract only what is necessary.

Furthermore, the project's diverse scope introduced us to the complexities of handling online requests, interpreting HTML structures, and displaying enhanced data via an easy-to-use interface. The significance of effective data handling, robust error control, and the demand for flexible solutions in the constantly evolving online world were underlined. In the end, this project functioned as a comprehensive look into the combination of frontend accessibility and backend scraping technologies, highlighting the ability of Python to extract insightful information from the web.

# **Bibliography**

|  |  |
| --- | --- |
| [1] | \*\*\*, "Python Home Page," [Online]. Available: https://www.python.org. [Accessed 05.11. 2023]. |
| [2] | V. Rossum, "Python3 Tutorial Documentation," 2000. [Online]. Available: https://docs.python.org/3/tutorial/. [Accessed 06.11.2023]. |
| [3] | W. McKinney, in *Python for Data Analysis*, O'Reilly Media, 2017, pp. 52-59. [Accessed 06.11.2023] |
| [4] | K. Reitz, "Requests: HTTP for Humans. Requests.," n.d. [Online]. Available: https://docs.python-requests.org/en/master/. [Accessed 06.11.2023]. |
| [5] | L. Richardson, "Beautiful Soup Documentation," n.d. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/. [Accessed 06.11.2023]. |
| [6] | Django, "Django Documentation," n.d., [Online]. Available: https://docs.djangoproject.com/. [Accessed 07.11.2023]. |
| [7] | K.-M. J. Holovaty A., "The Django Book: Version 2.0," https://djangobook.com, 2009. [Accessed 07.11.2023] |
| [8] | Django, "Django Packages," Django, n.d.. [Online]. Available: https://djangopackages.org. [Accessed 08.11.2023]. |
| [9] | T. Linus, "Git Source Code Management," Git, 2005. [Online]. Available: https://git-scm.com/. [Accessed 10.11.2023]. |
| [10] | Github, "Github Documentation," Github, n.d.. [Online]. Available: https://docs.github.com/. [Accessed 11.11.2023]. |