

Final Report: Graphical Crawler Application

Team Grus: Aaron Berns, Jacqueline Bonner, Hunter Schallhorn

Project Code: <https://github.com/anberns/grus>

Project Link: <https://graphicalwebcrawler.herokuapp.com/>

I. Introduction

Our graphical web crawler is a web application that offers the user multiple views of the different connections between sites on the web. Users choose a starting site and decide between a depth-first search in order to see just how far they can get from their starting site, or a breadth-first search, which will show them all of the different connections available from their starting site and all of the connections available from those connections. The connections are represented in graph form, with sites as circular nodes and directional arrows detailing the links between sites and order in which they were visited. After the search is complete, users may choose two different views of the completed graphs. One view better illustrates the path taken during the search by only showing the sites that were visited and the order in which they were visited. The other view visually displays the shared connections between the visited sites by showing all of their possible links. Both views provide information to the user about the total number of sites visited, the total number of sites found, and the number of links found between those sites. Our app provides the option of a keyword or phrase search, which stops the crawl if the keyword is found and indicates the site on which it was found in the graph. Graphs nodes can be hovered over to display the url they represent and can be clicked on to open their url in a new browser window. A more detailed explanation of our implementation follows.

Description of Software and Systems

- A. UI:** Our UI consists of a Bootstrap-styled, Jinja-enhanced html view that displays the title of our app and instructions for its use in the header, a form with entries for the starting site of the crawl, an optional keyword to search for during the crawl, a link limit to create a bound on the crawl's size and a radio option for choosing a breadth-first or depth-first approach.

The form includes built-in Bootstrap input validation that checks for blank entries in the starting site and limit fields. The starting site is validated by the server using the Requests Python library once the form is submitted and the user is prompted to re-enter a valid URL if the validation fails.

To the right of the form on wide-screen devices such as a laptop, or above the form on smaller devices, a list of previous crawls populates if any have been recently completed. This list includes what search type was performed, the limit, and also indicates whether a keyword was searched for. If a keyword was entered and not found, the word or words will be shown in black. If the keyword was found, it will be displayed in green. Each previous crawl can be clicked on to display its graph.

- B. *Data Visualizer:*** The data visualizer portion of the web crawler consists of a header with crawl information and navigation buttons, and a d3js force-directed graph. As the crawl is taking place, the message line displays the search type, starting URL, keyword if entered and a triple dot indicator that the crawl is actively taking place. Below the message line, a line of website stats tells the user the number of unique sites included in the crawl, the number of actual sites visited by the crawler and the number of links between all sites in the crawl. These stats are updated in real time as the crawl progresses and remain with their end values once it finishes.

Below the stat line, a pair of buttons give the user the option of viewing or hiding the titles of the sites being graphed, with the titles being hidden as a default. In the main portion of the page the graph grows in real time as the crawler function sends link data for each visited site over a websocket connection. Each site is a different color and is represented by a circular node. Arrows connect each site node, indicating the order of visitation in a depth-first search and the parent-child relationship in a breadth-first search.

Upon completion, the message line in the header updates to remove the triple dot animation and replaces it with the total crawl time in seconds. New buttons are added including an option to begin a new search and the option to show and hide the unvisited sites that are included in the crawl. If a keyword or phrase was chosen and found during the crawl, the site on

which it was found will pulse. All site nodes can be dragged and repositioned with the links automatically repositioning themselves to preserve their connections. The user can zoom in and out on the graph by sliding two fingers up and down on the trackpad, using CTRL+scroll wheel on a mouse, or pinching to zoom on a mobile device. The graph as a whole can be repositioned in the window by clicking anywhere on it and dragging. Each site node displays its URL when hovered over and can be clicked on to open its URL in a new browser window.

- C. Server:** The server is written in Python and uses the [Flask](#) library to enhance its functionality. It uses [Gunicorn](#) as the deployment wsgi server and [Flask_Pymongo](#) to connect with the MongoDB database, which is hosted by mLab. It uses [Flask_sockets](#) to implement a websocket connection between the crawler and visualizer.

Each new user is given a randomized id number using Python's uuid library that is stored as a cookie in their browser using Flask's sessions module, and which is used to query a database for previous crawls saved under that number.

The server consists of four routes: index, submit, crawl and previous. The index route, '/', first checks the user's browser for a previously stored userid cookie. If one exists, the route queries the the database and loads the index page discussed in the UI section above, populating the 'previous crawls' section with the data returned from the query. If no userid value is found, one is created and saved in the user's browser as a cookie and the index page is loaded with a blank 'previous crawls' table.

The form on the index page posts its data to the '/submit' route. This route saves the form data in global and session variables after validating the posted URL as described in the UI section. Finally, the route renders the view which contains the visualizer, "show_data.html", passing the crawl options as arguments.

The '/crawl' route is utilized by the "show_data.html" view to open a websocket and launch the web crawler portion of the application. The crawl options are accessed from the session variables mentioned above, and inserted into the database to create a new crawl document. The

Flask_sockets library creates a websocket when this route is followed. The crawler function is then called and the websocket, the document id from the database insert mentioned above and the stored crawl options are passed as arguments. The crawler function then communicates directly with the javascript logic inside of the “show_data.html” view via the websocket to render the graph.

The ‘/previous’ route is utilized when a previous crawl is clicked on from the index page. This route queries the database for the stored crawl data and passes the entire crawl to the visualizer when rendering the “show_data.html” view.

- D. Crawler:** The crawler is also written in Python. The crawl function has been written to receive the information from the server as to the the URL to launch the search from, the type of search to be completed, a keyword (if provided), and the depth limit set by the user. The websocket and database information are also passed in. The crawler then instantiates either a BFS or DFS Spider object in order to perform the crawl.

Both the BFS and DFS Spiders begin their searches by checking if the site is both able to be visited and retrieved. First, the crawler looks at the robots.txt file for the base URL of the link it is crawling using the [robotparser](#) library. This step was taken in order to ensure our web crawler follows the Robot Exclusion Protocol, and does not crawl areas that are disallowed by the site’s owner. It then checks if the URL returns a valid status, using the [Requests](#) library. During this step, if any link is returned as either not crawlable or not able to be loaded, then that link is skipped in the crawl and never visited. The next link on the list will be chosen to crawl.

Assuming it passes both of the checks above, the crawler then parses the page using [BeautifulSoup](#) in order to find all of the links on the page and the title of the page. Once the page has been parsed, the search continues in one of two different ways, as discussed below.

The BFS starts by saving off information found during the the parsing in a dictionary format - the URL, title of the page, list of links available on that page, the current depth of that page, the parent of that page, and a flag to

indicate whether the keyword was found or not. That information is then updated on the database and sent to the websocket. Next, the crawler goes through all of the links that were found on that parent site and adds them to a queue of links to be visited, along with pertinent information about those URLs. At this step, each URL is first formatted to create absolute URLs from relative links and to help avoid repeated links by getting rid of queries and anchors. Each URL is then checked for validity using the [validators](#) library, for duplicate URLs, and for media extensions before being added to the queue. Once these child links have been added to the queue, the crawl loops back around to repeat until it either has an empty queue, has found the keyword, it reaches the user-specified depth limit, or it reaches a maximum link limit of 35,000.

The DFS starts off by saving off the information about the current page, in the same way as the BFS and sending it to the database and websocket. However, instead of creating a queue of all of the links found on each parent page, it picks a next link from that list to follow at random. Like in the BFS, the link is first validated before being tagged as the next link to follow. This loops back around until it either has no next link to follow, the keyword has been found, it reaches the depth limit set by the user, or it reaches the maximum limit of 35,000 links.

III. Libraries, Languages, Tools, and Technologies

A. Client-Side:

1. Javascript is the language used to write scripts for the front end to make the UI and visualization more interactive utilizing the libraries mentioned above.
2. HTML is the language used to set up the layout of the elements on the front end.
3. CSS is the language used to style elements on the front end. With CSS, we are able to give context to elements with color, size, and placement.

4. Jinja is the default templating engine for flask. It allows data to be passed, manipulated and displayed on the front end of the webapp. Jinja is mostly used to determine if an error was being passed from the server, to display an empty list when the user has no previous crawls, and to reverse, format, and display the previous crawls when they are present.
5. JQuery is a Javascript library that is used to manipulate HTML elements on both the UI and visualization.
6. Bootstrap is another library built on top of jquery. It is used to make the front GUI responsive and consistently styled. If the user is on a smaller screen or changes the size of their browser, the layout automatically resizes and readjusts the layout of components to best fit the screen.
7. D3.js is a Javascript library that helps bind data to simulations and SVG elements. D3 is the driving force behind the majority of the visualization, binding data received from the server to the forces and visual elements of the visualization. Not only did d3 provide the graph simulation to automatically position where the nodes and links should go, but it also made manipulating and updating the SVG elements a breeze.
8. WebSockets are a web technology that allows the front end to communicate with the back end (and vice versa) without having to use ajax requests, long polling, or page refreshes. This allows us to display data from the crawler as it is encountered in real time at the crawler end. This frees up memory used by the crawler as it no longer has to store all its own data, having been passed off to the server and front end. This also proves to be a better experience for the end user as they no longer have to wait for a crawl to finish before seeing the data. This also allows us to get around request timeouts for longer crawls.

B. Server-Side:

1. Python3 is the language used in writing the server and the crawler. Python afforded us multiple libraries that were built to assist in web-crawling, formatting data, and websocket integration.

2. JSON is the notation used to pass larger amounts of data between the crawler, server, and front end and is also the notation MongoDB used to store data. The crawler sends JSON formatted messages to the front end to be parsed and formatted into the data d3js expected. The server passes JSON messages stored in the database to the front end for it to display previous crawls.
3. Flask is a microframework for Python. It is the server used to handle requests from the front end, routing, and event handling. Flask makes it easy to deal with data passed with the request, reading and setting data stored in the cookies, and passing data to the front end with the addition of Jinja templating.
4. Gunicorn is the deployment wsgi server used to serve our Flask application.
5. MongoDB is a noSQL database that stores data in json documents. This was the natural storage solution for our team as all of our data is already being passed and parsed as JSON messages.
6. Flask_sockets is a websocket library specifically for flask. It is used to create the other end of the socket connection to the front end in order to pass data from the crawler to the visualization in real time.
7. Heroku is used to host the server. With Heroku, we could set it up to automatically pull and deploy changes from our github master branch. Heroku also allowed us to attach a MongoDB instance to our dyno, to access logs for debugging, and to collaborate to build our product.
8. Robotparser is a Python library which was used to obtain information about whether or we are allowed access to fetch a certain path on the website that published the robots.txt file, by fetching that file and parsing it. While it does have a limitation in its inability to support sitemaps, we felt it was adequate for the scope of our program.
9. BeautifulSoup is a Python library used to parse the HTML DOM in order to find both the title and the links available on the page. BeautifulSoup was also used to search for the user provided keyword on the page.
10. Requests is a Python library used to send HTTP requests in order to retrieve the website for parsing. In addition, we were able to use

this library in order to check the response status code to check for valid starting urls and to check for bad requests during the crawl in order to avoid errors when attempting to load each page for parsing.

11. Validators was used to do an initial check of the formatting of each URL being entered into the crawl, to ensure it is in a valid URL format.

C. Other Tools:

1. Git is a version control system. Git allowed us to all work at the same time by pulling and pushing code changes, rewinding to old versions of code, and host multiple instances of our app on separate branches, specifically so we could make changes while the midpoint check was being graded.
2. Github is web based hosting for git repositories. This is where main repository for our project resided, rather than on one of the team member's personal computers.

IV. Testing

Both integration and regression testing was completed during each step of implementation. In order to accomplish this level of testing, we chose to create a separate testing environment linked to a testing branch in git. This allowed us to test potential fixes while keeping our master version live. The person testing would document their progress in git, which would post live updates and deployments to our chat. They would then turn over the testing branch to others for confirmation before merging the changes to the master version.

When bugs were found, we communicated them in our shared chat and worked together to brainstorm possible reasons for them. Then, one person would try and track down those bugs using a combination of Chrome's DevTools, Heroku's logs and code inspection.

Initial testing was done using a small site we created, in order to explore a known structure for expected results. Once we moved past initial stages, we pushed our crawler by using much larger websites such as www.google.com, www.apple.com, and oregonstate.edu.

Additional stress testing was completed once all features had been finalized in order to determine the optimal value for the link limit in order to prevent freezes. During this time, the decision was made to refactor the Javascript code in order to make it more efficient and really push the limit for what the browser could handle. While the website will not crash until after 40,000 links found, it was decided by the team that the slowdown was too much after 35,000, as the user would notice significant delays at that point.

V. Deviations From the Original Plan

Originally, we had not considered implementing a hard limit on the size of the dataset created by the crawl. However, when testing larger breadth first searches, the limitations of the client-side parsing over 35,000 links became very apparent performance-wise. To keep the page from slowing down to a point where it would be very noticeable by the user or crashing, the crawler will stop and close the websocket early if it reaches over 35,000 found links. This early stop could affect its ability to find a keyword or reach the user-supplied depth limit. However, we felt it was a necessary change in order for us to be able to display the additional information on connectivity that we included.

Our group had also not planned for all the ways we were going to allow the user to manipulate how the graph was displayed. However, we decided to allow the user to toggle the page titles on and off, as well as opt to display the unvisited sites or not. Both these additions provided interesting and different ways to display the graph to the user, depending on their preference.

Various UI changes were also made to increase the readability of the graph. This included making the nodes and links a color other than black, to ensure the titles could be properly read, indicating depth with the color, and adding an animation to the found site to ensure it is easily seen by the user.

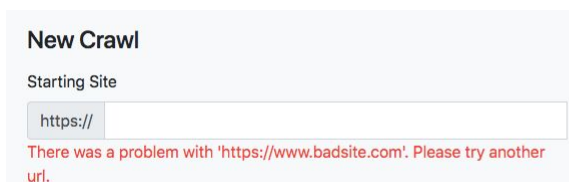
Another deviation from our plan, included the addition of an attempt to adhere to politeness protocols by obeying robots.txt files at each site before attempting to fetch and parse the page. We decided that this was an important addition after doing more research on well-designed web crawlers. During this research, we came across several articles on the importance of adhering to these protocols

in order to prevent our crawler from being banned or falling into various types of robot traps.

Last, we decided to remove threading from the crawl level. This was a decision made based on the addition of the websockets, which we felt eliminated the need for it.

VI. User Instructions

1. Navigate to <https://graphicalwebcrawler.herokuapp.com/>
2. Enter a valid starting URL. If, upon submission, the field is left blank or an invalid URL is entered, you will be prompted to re-enter a valid URL.



The screenshot shows a web form titled "New Crawl". Below the title is a label "Starting Site" followed by a text input field. The input field contains "https://" and is followed by a red error message: "There was a problem with 'https://www.badsite.com'. Please try another url." The error message is in red text and is positioned below the input field.

3. If you wish, you may enter a keyword or phrase. This field is not required to begin a crawl. The keyword will cause your crawl to end early (before your chosen depth limit), if found.
4. Enter a limit. A limit of at least 1 is required. This limit will indicate the maximum number of links to visit during a depth-first search, and the maximum distance from the original site it will reach in a breadth-first search. Your crawl will end when it reaches your limit, or it finds a total of over 35,000 connections. If, upon submission, this field is left blank, you will be prompted to enter this field.
5. Choose a search type. This will change how the crawl is completed and the resulting graph. Depth-first is set as the default.

6. Click 'Crawl'.
7. The visualizer will begin and the graph will grow with each site visited. The stats line will update as the crawl progresses. Site titles can be toggled on and off while the graph is growing.
8. Click on "Show Titles" to see the site titles, then on "Hide Titles" to hide them.
9. Once the graph finishes building, you will be able to click on the "Show Unvisited" and "Hide Visited" options. The "Show Unvisited" option allows you to view sites that were found during the crawl, but were not visited for various reasons. This will also change the orientation of the visited sites on the graph, due to any shared connections between visited sites. Click on "Hide Unvisited" to hide the sites that were found, but not crawled.

Graphical Web Crawler - Grus

Breadth-first search on 'https://www.apple.com' completed in 210.30 seconds

Unique Sites: 3301 - Visited Sites: 424 - Links: 35009

[New Crawl](#)[Hide Unvisited](#)[Show Unvisited](#)[Hide Titles](#)[Show Titles](#)

10. You may also both manipulate and explore the graph you see.
 - a. Click anywhere on the graph in order to move it within the browser window.
 - b. Click on a node and drag it in order to move that node and any children it has.
 - c. Scroll upwards with two fingers to zoom in. Scroll downwards with two fingers to zoom out.
 - d. Hover your mouse over a link to see its URL display.
 - e. Click on a link to open it in a new window.
11. If you wish to start a crawl with new options, click on “New Crawl”.
12. The home page will load and the crawl you just completed will appear in the “Previous Crawls” section. In this section, you will be able to see the starting site you entered, the type of search you completed, and the depth limit of that search. In addition, if you chose to enter a keyword, it will also appear here. If that keyword was found during your crawl, it will be shown in green. Click on the URL of the previous crawl and the prior completed graph will load.

VII. Screenshots

Figure 1. Start of a depth first search on the New York Times website with the keyword 'Oregon' and with no previous searches stored on the user cookie.

Graphical Web Crawler - Grus

Performing a depth-first search on 'https://www.nytimes.com' for the word 'Oregon '...

Unique Sites: 324 - Visited Sites: 4 - Links: 383

Hide Titles Show Titles

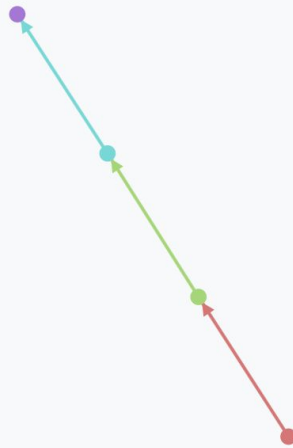


Figure 2. Mid-crawl with titles hidden. The statistics on sites found, sites visited, and links found update as the graph updates with each new site visited.

Figure 3. Complete depth-first search with optional titles shown. The keyword 'Oregon' was not found, so the crawl continued until the limit of 15 was reached.

Figure 4. The 'Show Unvisited' option reveals further connections between the visited sites and their unvisited links.

Figure 5. Start of a breadth first search on Apple's homepage with no keyword and the previous search listed under 'Previous Crawls'.

Graphical Web Crawler - Grus

Breadth-first search on 'https://www.apple.com' completed in 248.18 seconds

Unique Sites: 3445 - Visited Sites: 425 - Links: 35081

New Crawl

Hide Unvisited

Show Unvisited

Hide Titles

Show Titles

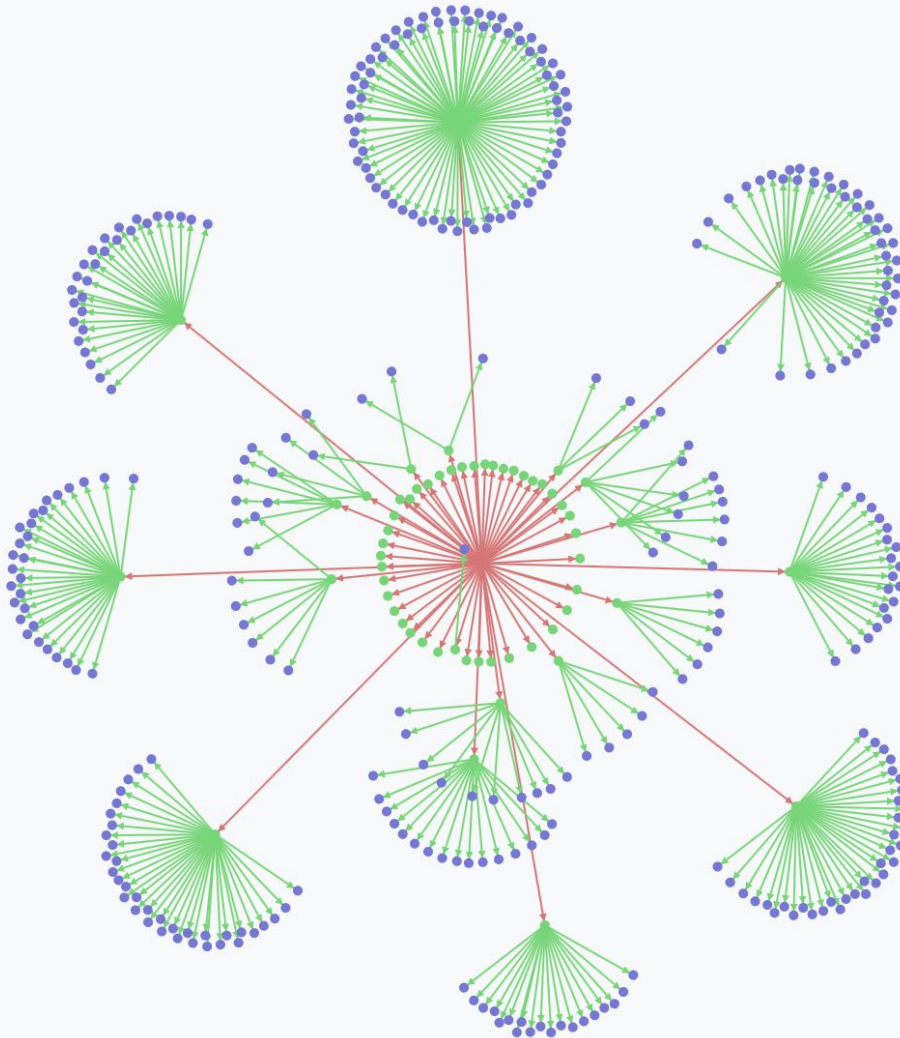


Figure 6. Results of a breadth first search on Apple's homepage. Each tier of links is given a different color. The green second tier of sites is incomplete as the total link count has surpassed 35,000, ending the crawl.

Graphical Web Crawler - Grus

Breadth-first search on 'https://www.apple.com' for the word 'keyboard' completed in 1.35 seconds
Keyword 'keyboard' found.

Unique Sites: 151 - Visited Sites: 5 - Links: 329

[New Crawl](#) [Hide Unvisited](#) [Show Unvisited](#) [Hide Titles](#) [Show Titles](#)

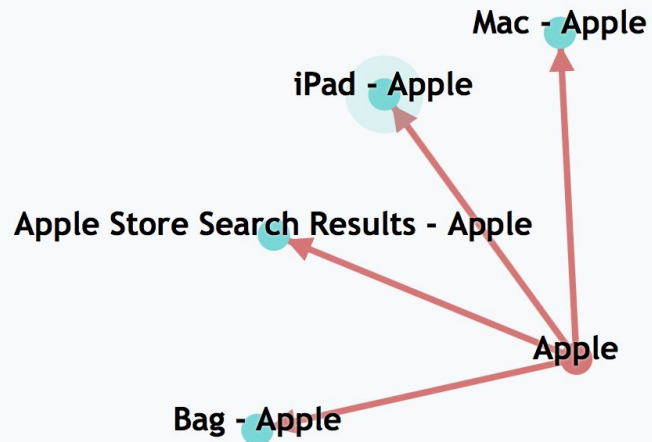


Figure 7. The same crawl as above but with the keyword 'keyboard'. The keyword is found on Apple's iPad page, as indicated by the pulsing node. Since the keyword was found, the crawl ends before the limit of 2 is reached.

Figure 8. Hovering over a node reveals the URL for the site represented by that node. Here, hovering over the pulsing node reveals the site on which the keyword was found.

VIII. Team Member Accomplishments

Aaron Berns researched and implemented the server, database interactions and websocket functionality, and setup the web hosting and database.

Jacqueline Bonner researched and implemented the crawler and performed unit and integration testing for the application.

Hunter Schallhorn researched and implemented the UI and visualizer, and performed unit and integration testing for the application.

IX. Conclusion

The process of planning, building and deploying our web crawler application allowed each group member to gain countless insights into the software development process. Tackling a client-defined application that required several components previously unknown to our group members highlighted the necessity of thorough planning and research, iterative development, integrated testing and developer collaboration in the software development process.

Along with giving us invaluable software development experience, this project made us aware of the power and pitfalls of web applications. Countless developers have created tools and libraries to address all aspects of a web application. Many of these tools ease the burden of bringing a concept to life and are indispensable to the web development field. Each has their unique drawbacks and eccentricities, however, and we learned over the course of building our app how important it is to fully understand the value that each can add as well as the restrictions that each can impose on other aspects of a project. An example of this is the `Flask_sockets` library. It proved easy to implement, functioned robustly, and worked seamlessly with our host, Heroku. This was something that other Python socket libraries could not claim in their list of benefits. A major drawback of this library, however, is that it removes the user's ability to specify a number of worker threads for each app instance, making threading within the app impossible without forking the source code and creating a personalized version of the library. We had similar experiences of awe and frustration when attempting to parse sites with Javascript generated content in working with Selenium and Chromedriver. This was a feature that we never ended up getting to work due to the timeout limitations set by Heroku. These experiences further highlighted the importance of exhaustive research and planning when choosing the technologies to integrate into an application.

Overall, building our application was incredibly educational and rewarding. We feel that we have built a user-friendly, feature-packed, and technically complex web crawler that highlights some of the often overlooked complexity and connectedness of the web. We hope you enjoy using it!

X. References

<https://prateekvjoshi.com/2016/03/08/how-to-create-a-web-server-in-python-using-flask/>

<http://flask.pocoo.org/>

http://www.bogotobogo.com/python/MongoDB_PyMongo/python_MongoDB_RESTAPI_with_Flask.php

<https://progblog.io/How-to-deploy-a-Flask-App-to-Heroku/>

<http://api.mongodb.com/python/current/tutorial.html>

<https://stackoverflow.com/questions/1210458/how-can-i-generate-a-unique-id-in-python>

<https://github.com/heroku-python/flask-sockets>

<http://docs.gunicorn.org/en/stable/run.html>

<https://codepen.io/xwildeyes/pen/KpqVzN>

<http://flask-pymongo.readthedocs.io/en/latest/>

<https://github.com/heroku/heroku-buildpack-google-chrome>

<https://stackoverflow.com/questions/41059144/running-chromedriver-with-python-selenium-on-heroku>

<https://pythonspot.com/category/selenium/>

<https://stackoverflow.com/questions/26745519/converting-dictionary-to-json-in-python>

<https://docs.python.org/3.1/library/uuid.html>

<http://www.pythonforbeginners.com/beautifulsoup/beautifulsoup-4-python>

<https://beautiful-soup-4.readthedocs.io/en/latest/>

<http://validators.readthedocs.io/en/latest/>

<http://docs.python-requests.org/en/master/>

<https://docs.python.org/3/library/re.html>

<http://www.robotstxt.org/orig.html>

<http://blog.mischel.com/2011/12/20/writing-a-web-crawler-politeness/>