Automated Crisis Events Collection Builder

Final Project Report

Team 8

Client: Dr. Mohamed Farag

Brian Hays, Mitchel Rifae, Parsa Nikpour, Trevor Kappauf, Alex Zhang

CS 4624

November 11, 2023

Table of Contents

1. **Abstract**
2. **Introduction**
   - 2.1. Project Summary
   - 2.2. Client
   - 2.3. Deliverables
3. **Requirements**
   - 3.1. User Customized Run
   - 3.2. Run Crawl on Input File
   - 3.3. Descriptive Statistics
   - 3.4. Export Statistics
   - 3.5. Filter by Specific Crisis Events
   - 3.6. User Authentication
4. **Design**
5. **Implementation**
   - 5.1. backend.py
   - 5.2. App.js
   - 5.3. Home.jsx
   - 5.4. CrawlInitializer.jsx
   - 5.5. CrawlCardHolder.jsx
   - 5.6. CrawlCard.jsx
   - 5.7. Crawl.jsx
   - 5.8. TreeVisualization.jsx
   - 5.9. Banner.jsx
6. **Research**
7. **Testing**
8. **Lessons Learned**
   - 8.1. Timeline
   - 8.2. Problems
   - 8.3. Solutions
   - 8.4. Future Work
9. **Acknowledgements**
10. **References**
11. **USER MANUAL**
12. **DEVELOPER MANUAL**

List of Figures

Table 1 Team Development / Planning Increment Timeline

Table 2 List of URLThing Class Fields and Use

Figure 1 Crawl Card Filtering

***Figure* 2** Data being Mapped

Figure 3 API Request Format

Figure 1 Crawler Diagram

Figure 2 read_urls_from_file method

Figure 3 read_keywords_from_file method

Figure 4 React Router Setup

Figure 5 Link functionality

Figure 4 URLThing Class

Figure 5 __lt__ Comparison for max heap

Figure 6 __str__ method

Figure 7 toJSON method

Figure 8 Methods to add/remove from list

Figure 9 Predict method

Figure 10 Pop/Push methods for max heap

Figure 11 Downloading htmls to Firebase Storage code

Figure 12 download_from_storage method

Figure 13 run_to_db method

Figure 14 build_hierarchy method

1. Abstract

In the contemporary digital landscape, access to timely and relevant information during crisis events is crucial for effective decision-making and response coordination. This project addresses the need for a specialized web application equipped with a sophisticated crawler system to streamline the process of collecting pertinent information related to a user-specified crisis event.

The inherent challenge lies in the vast and dynamic nature of online content, where identifying and extracting valuable data from a multitude of sources can be overwhelming. This project aims to empower users by allowing them to input a list of newline-delimited URLs associated with the crisis at hand. The embedded crawler software then systematically traverses these URLs, extracting additional outgoing links for further exploration. Afterwards, the contents of each outgoing URL is then run through a predict function, which evaluates the relevance of each URL based on a scoring system ranging from 0 to 1. This scoring mechanism serves as a critical filter, ensuring that the collected web pages are not only related to the specified crisis event but also possess a significant degree of pertinence. We allow the user to set these thresholds, which enhances the efficiency of information retrieval by prioritizing content most likely to be valuable to the user's needs.

Throughout the crawling process, our system tracks a range of statistics, including individual website domains, the origin of each child URL, and the average score assigned to each domain. To provide users with a comprehensive and visually intuitive experience, our user interface leverages React and D3 to display these statistics effectively.

Moreover, to enhance user engagement and customization, our platform allows users to create individual accounts. This feature not only provides a personalized experience but also grants users access to a historical record of every crawl they have executed. Users are further empowered with the ability to effortlessly export or delete any of their previous crawls based on their preferences.

In terms of deliverables, our project commits to providing fully developed code encompassing both frontend and backend components. Complementing this, we will furnish comprehensive user and developer manuals, facilitating seamless continuity for future students or developers who may build upon our work. Additionally, our final deliverables include a detailed report and a

compelling presentation, serving the dual purpose of showcasing our team's progress across various project stages and providing insights into the functionalities and outcomes achieved.

2. Introduction

2.1 Project Summary

The objective of this project is to build a web application that enables users to run crawler software in the backend. This crawler software will collect web pages related to a user provided crisis event. The user input will be in the form of a list of new line delimited URLs. The crawler will proceed to traverse each provided URL, gather additional outgoing URLs, and if the predict function deems them valid, it will iterate through these outgoing links. The predict function assigns a score to each URL on a scale of 0 to 1, thereby gauging the web page's relevance to the user's crisis event. Finally, the crawler will output a collection of input-set derived webpages.

2.2 Client

Our client is Dr. Mohamed Farag who is a research associate in the Center for Sustainable Mobility (CSM). Our client needs an application that creates a collection of web pages that are related to a specified crisis event.

2.3 Deliverables

To achieve our goals and objectives for this project we plan to deliver the following:

1. A Web Application capable of running such crawls, accompanied by its source code.
2. User manuals of the web interface and the web crawler.
3. Developer manuals of the web interface and the web crawler. This will allow for future developers to continue development with ease.

3. Requirements

We decided that our web application will have the following requirements: user customized run, run crawl on input file, descriptive statistics of crawls, export statistics, filter by specific crisis events.

3.1 User Customized Run

The user will be able to customize every crawl they run. In detail, each user will be able to give each crawl a custom name, set thresholds, as well as a hard count. Significantly, the thresholds and hard count parameters will affect the web pages returned from the web crawl.

3.2 Run Crawl on Input File

Each user will be able to input a file of user-selected URLs along with a text file with custom keywords that will be used to determine the score of each URL. Our backend will then parse each URL from the given input file and run a crawl on each of them.

3.3 Descriptive Statistics

Throughout the crawling process our backend software will keep track of statistics about the crawl. For instance, we will keep track of domains, the origin of each child URL, and the average score assigned to each domain. After the crawl finishes the user will be able to view all statistics that we derived.

3.4 Export Statistics

After a crawl has been completed and our statistics have been derived, the user will have the ability to receive the hierarchy tree of all the domains along with the average score of each domain as a JSONArray. This feature ensures that users can seamlessly integrate the obtained insights into their workflows, enhancing the overall utility of the platform.

3.5 Filter by Specific Crisis Events

The user will have the ability to filter through all their crawls, to find a specific collection.

3.6 User Authentication

Each user will create a unique account. This personalized account not only adds a layer of authentication but also provides users with the capability to save and revisit each crawl.

4. Design

When developing our web application, we organized our tasks into two key groups: frontend and backend. We began the project by giving priority to the Python backend, recognizing it as the area with the most uncertainties. During the design phase, we experimented with various Python libraries to identify the most programmer friendly option for web scraping. Simultaneously, we conducted research to determine the most suitable database that both the backend and frontend teams could agree upon.

Our decision to focus on the backend first was reinforced by the frontend team's already well-defined understanding of their implementation plan. This strategic approach allowed us to efficiently allocate resources and address potential challenges early in the development process.

This way of organizing our work helped us stay focused and work closely with the client. The client's early input was crucial in giving us useful advice, especially about setting up the Python backend. This teamwork made sure we dealt with possible issues early on, making decisions wisely and keeping the project running smoothly.

Our project's success is evident in meeting deadlines and delivering the web application on time. Planning tasks carefully and communicating with each other set the stage for a well-coordinated and successful development process.

5. Implementation

We implemented this project by using a combination of python, flask, react, css, d3, and html. Using python as the backend to communicate with our firebase database as well as our flask api. Then using react, d3, css, and html for the user interface.

5.1 backend.py

We've consolidated our Python code into a single file, backend.py, for streamlined functionality. This approach offers the advantage of using global variables that adapt to user preferences and crawl thresholds, while also keeping track of essential statistics during the crawl.

Our backend.py interacts seamlessly with the frontend through the Flask API, employing the following routes: "/delete_firebase_entry," "/get_downloaded_htmls," "/get_user_crawls," and "/scrape_and_save." The pivotal route is "/scrape_and_save," functioning as our main method initiated by the frontend to commence a crawl. The other routes serve specific purposes: "/delete_firebase_entry" deletes a user crawl, "/get_downloaded_htmls" exports statistics, and "/get_user_crawls" populates the user's crawls upon login.

These routes connect to functions that, in turn, call various helper methods to execute a successful crawl and relay essential information to the frontend. This modular and efficient design ensures a smooth flow of data and operations between the backend and frontend components.

5.2 app.js

The app.js file contains the logic we require to handle user authentication via. firebase, which works in tandem with the firebase-config.js file, generated through our unique application's auth information. **[insert a link about firebase auth generation here]** Due to the nature of the authentication, we have access to a user object, which is populated asynchronously with User-initiated events. This consists of signing in, registering as a user, or signing out. Once this object is populated, it is mounted to the [user, setUser] state object. Due to security, and other frontend framework issues that would derive from having an unauthenticated user accessing the home page of our application **[see section 5.3]** our App.js handles dynamic rendering. In the case that the user object is populated, and not null, we dynamically render the Home page for the user, and send the user object to Home.jsx as a catchable prop for later usage. In all other cases (which is a state in which the user object is null) we dynamically render the user authentication menu, that being - the sign in / registration content.

5.3 Home.jsx

The Home.jsx file is rather straightforward, in that it acts as a container for two components, one being the Banner **[see section 5.10]** and the other being CrawlInitializer. **[see section 5.4]** The

Home page acts as a mediator between App.js, and the CrawlInitializer & Banner, so that when App.js determines the user object is populated, and Home.jsx should be rendered, CrawlInitializer & Banner can pass in the user object components they require for their logic.

5.4 CrawlInitializer.jsx

CrawlInitializer is the hub of all frontend logic, and all of the backend communication logic.

5.4.1 *CRUD Operations*

CrawlInitializer is the Hub for all of our backend communication logic. On load, this component makes a request to the flask server through the **"/get_user_crawls"** route, and retrieves all historical crawls run by the authenticated user. In addition, this component supports the insertion of new crawls, and deletion of historical crawls. For more information on the process for running a new crawl, **see section 5.4.3.** Regarding deletion, each card in the visual UI is accompanied by a "Delete Crawl" button. Due to the explicit nature of the button in the UI, there is no user-error prevention. Simply clicking the "Delete Crawl" button deletes the entry via. The flask route **"/delete_firebase_entry"** which appends the crawl name of the card being deleted, for the python to check against in the database.

5.4.2 *Child Component Props*

The CrawlInitializer.jsx file has one key child component, that being the CrawlCardHolder component. **[See section 5.5 for more information].** Regarding the props that are sent in, there are 3 pivotal ones. The first, is the data retrieved from the CRUD Operations **[see section 5.4.1]**

5.4.3 *Running a New Crawl*

To run a new crawl, a user must engage with the interface with the "To Run a Crawl…" header at the top, which provides a Drag & Drop menu, for the user to provide a file of new line delimited URLs, for a crawl to be run on. Once a file is provisioned, two menu options will appear. One, will allow a user to remove a file, and reprovision one. The other is an option to open a menu to set constraints. A user must set 2 key constraints before a crawl can be run. Those being, a Crawl name, and another file of **comma delimited keywords** to check each URL in the original file against, to generate a valid predict algorithm for the given crawl. Beyond this, all constraints are optional - and in the case that they are left alone, the displayed values are left as their defaults for

the subsequent crawl. Once a user has provisioned **at minimum** the two key constraints, a new menu item will open that enables the user to run a crawl. Again, due to the explicit nature of the text accompanying this menu option, there is no additional check for user error. Clicking this button will initiate a crawl immediately. Upon selection of this button, a skeleton CSS spinner will mount, and unmount, succinctly with the initialization & completion of a run via. The flask server. Once this crawl completes, another CRUD operation **[see 5.4.1]** will be performed to retrieve all database crawl records the user is tied to - of which the newly run crawl will be apart - simply scrolling down will display the cards in their default state, including the new crawl run by the user. To speak briefly on generation strategy, we leverage the ***uniqid*** library to generate a new, unique id for each crawl as its database identifier. In this way, we are able to accomplish CRUD operations such as deletion on a card-by-card (or rather crawl-by-crawl) basis.

5.5 CrawlCardHolder.jsx

CrawlCardHolder.jsx acts as a mechanism for mapping the data retrieved from the CRUD operations **[see 5.4.1]** into Cards for the visual UI. **[see 5.6]**. In addition to this, CrawlCardHolder.jsx supports the functionality of filtering CrawlCard.jsx components by their Crawl Names. If you recall, users are mandated to provision a Crawl Name prior to initialization of a new crawl. This being the case, all Crawl Cards will have a name - though uniqueness is not mandated, as an underlying unique identifier already exists to satisfy the requirement for database entries, levying the ***uniqid*** library through npm **[as explained in 5.4.3].** The filter works by leveraging a standard input box, with its value & onchange functions routed to a state. When the input box registers a change in state, it filters the data retrieved from CrawlInitializer **[see 5.4.2]** to only contain card/crawl entries that consist of the filter's text somewhere in the overall Crawl Name string. This voids any need for wildcard references, or correct placement of words chronologically, as the search is all-intensive - as long as the substring in the filter exists in n >= 1 card(s), the filter will return said filtered card(s). As we mentioned, this JSX component acts as a mapping mechanism. In such fashion, once the data has correctly mounted from the parent component's CRUD operations, the react simply maps over each entry in the passed data array, and converts each of them into a new "CrawlCard" object. Due to the nature of the JSX structure in the component file, this generates a collection of cards all sub-contained in the

CrawlCardHolder.jsx file, with each being passed the object entries of each map index **[see CrawlCard.jsx]** for later use.

5.6 CrawlCard.jsx

CrawlCard.jsx acts as a container for a single entry in the overall data array retrieved in CrawlInitializer, as passed in by CrawlCardHolder **[see 5.5 & 5.4.1]** - as well as possessing its own unique delete CRUD operation, for its own crawl instance. Pivoting off of the understanding that each CrawlCard object, is a visual representation of each entry in the data array - and thus contains all of the data for that given array entry - we once again use the data retrieved from the CrawlInitalizer, to formulate a visual UI component - and also a navigation strategy for expansive information on a unique crawl. First, for the method by which we visualize the array entry - we simply utilize the **crawl name, crawl date, and seed urls** for the key visualization. This is accomplished by direct insertion of the JS variables into the html text. We decided, for simplicity among extensive crawls & shorter crawls, to only list the first 5 seed urls in the original crawl constraint text document. **[see section 5.4.3]**. The key logic for this component lies in the wrapping of each card in a Link component, and the presence of a button to fire the delete CRUD operation, one layer higher in the z-index than said Link component. In order to provide more concise data on each Crawl, but not overburden the user with information on the landing screen - we opted for the strategy of dynamically paginating crawl information out, per each crawl existing in the database. To accomplish this, we simply leveraged **react-router-dom [2]** and passed all relevant data to a dynamic route, using the ***uniqid*** as the identifier by which we differentiated crawl pages from one another. **[see section 5.4.3]**. Due to the nature of the uniqid library, users will not be able to guess / estimate identifiers for crawls that are not their own. This is the assumption through which we assume sessioned data carries over between pages - that being, the impossibility of brute forcing a unique identifier generation library. In terms of user engagement, text built into the card prompts the user to “click the card” for statistics information. That being the case, simply engaging with the card at any location **[besides the delete button, obviously]** will allow the user to navigate to that crawl’s unique Crawl.jsx page, through the Link component. The Link component sends all data for statistics, those being **displaySeeds, stats tree, id, crawlName, and email** to the Crawl.jsx page as well **[see section 5.7].** Returning to the delete CRUD operation each Crawl Card possesses, the frontend - once the

delete button has been clicked by the user (once again, due to the explicit nature of the button's text, there is no second check against user error) a request to the flask server, with the Crawl ID appended as a formData object, is made for deletion. Once this deletion is accomplished, the CrawlCard.jsx component leverages its parent method passed in via. React props. **[see section 5.4.2]** this function has been passed down hereditarily all the way from CrawlInitalizer, so that when a child deletes a crawl, the necessary "get" request that exists in CrawlInitializer **[see section 5.4.1]** can be refired from the child component. This function call is propagated up to the parent, as its existence is scoped to the parent component, and being called from the Crawl Card. This is done for efficiency, and to consolidate widespread API calls in one function, when they can be.

5.7 Crawl.jsx

The Crawl.jsx component is rather straightforward. Since the page exists dynamically, and each route determines the source content through which Crawl.jsx displays it's information **[see 5.6 regarding why that is]** Crawl.jsx acts as a hub for all data visualization strategies, and downloadable content (such as the stats JSON, and HTML pages downloaded during the crawl). In addition to the downloadable content, we have two key visualization components. One will be explored at length in **section 5.8**, which is TreeVisualization.jsx . The other is more streamlined, that being the **MUI BarChart [6]** component. Due to the nature of the firebase storage, and what is valid in terms of structure, what the Crawl.jsx file has access to, is a JSON object that has been converted & stored in string format. This being the case, Crawl.jsx leverages a two step process, one - being the JSON.parse function, and the other being the Object.keys function. These two functions, in tandem, generate an object structure that the MUI library can correctly leverage to generate bar charts. The code is very explicit, but at a high level, the data component determining the parameters of the bar graph, takes in this newly parsed data, **termed "websites"** and maps it holistically into just its average URL score. This "average score" is coined & computed via the flask server, so the frontend simply needs to use its existence for the bar chart. For a more concise explanation of the structure of the **stats** and now **websites** objects, the following is true: each domain found in the history of the crawl, has a count of the amount of times it appeared, an array of all scores across those appearances, and then a composite average taken over that array. All of this data is computed by the backend, and we simply map each

domain entry to only be its average score, utilizing the domain name for the xAxis, by not mapping it whatsoever. Again, the code is rather explicit, but the differentiation between only using **websites** and using **websites mapped to be its avg_score component** is what allows the bar chart to generate its x vs. y relationship visually for all domains retrieved in the crawl.

5.8 TreeVisualization.jsx

As aforementioned, the TreeVisualization.jsx component leveraged the **react-d3-tree [5]** library, to generate a user-interactable tree consisting of the hierarchical tree structure generated by the crawl. To accomplish this, Crawl.jsx sends its **tree** parameter **[see section 5.7]** to the TreeVisualization.jsx component. Once here, due to the nature of crawls having n >= 1 seed URLs at the outset, we make the decision to uniformly instantiate & insert a dummy root seed URL at the top of the hierarchy. This way, regardless of the amount of seed URLs present in the initial constraints **[see section 5.4.3]** we are able to pass the data structure to the **react-d3-tree [5]** library as a true tree structure in JSON format. In addition to this, some basic data cleanup is necessary, in order to refine the several-component-object entries of our tree structure, into one that only possesses two, those being **name (for displaying text per node)** and **children (for containing all subsequent nodes in the hierarchy, with the given node as it's parent)**. This is accomplished by passing our newly formatted data (with the dummy root prepended) to a function that leverages the currently existing JSON object parameters, and compiles the relevant ones into a "name" parameter that contains the following:

Domain name, **Average Score, Amount of Children, Average Children Score**

This is accomplished by formatting the URL, and retrieving the base URL domain, formatting the pre-existing "Avg_score" parameter to a fixed length, and retrieving the child lengths & average composite scores recursively, based on the pre-existing "children" parameter the passed **tree** object contains. Once this data cleanup is complete, we have a hierarchical, node-based JSON object structure, where each node is a key-value pair of name-children. With this being the case, we simply pass this object to the **react-d3-tree [5]** Tree component. There is some accompanying logic regarding scaling, path curvature, zoom maximization, and CSS classes for visualization - but all of these can be found in more extensive detail through the documentation found here: **[link to react-d3-tree documentation]**

5.9 Banner.jsx

For more information on the nature of the props being passed into Banner.jsx, and where they derive from - please reference **sections 5.2, 5.3, and 5.6.** The banner simply serves as a UI accompaniment and visual strategy to make the transition from nothing to the Crawl-related components smoother from a UI/UX perspective. The aforementioned props are inserted into the "Welcome, {user}" statement, which either takes the form of Guest (from App.js) or the email of the given user (from Home.jsx and CrawlCard.jsx/Crawl.jsx). In addition, a unique description per each of the aforementioned JSX components is generated, providing specific instructions / accompanying dialogue for the user to understand where they are, and what functionality is present on the page, in addition to instructions on how to navigate.

6. Research

For this project we needed to research a few different API's and python libraries in order to arrive at the specific web scraper software that would best suit our web app. One of the first examples that we began to look at was the scrapy python library. Scrapy is very popular and has many different applications and tools for the programmer's convenience. However, we found this trait to be to the detriment of our program. This is because our program only required a small fraction of what the scrapy API provided meaning that many of the tools and features of scrapy would be extraneous to our project.

Pivoting over to the frontend, we tested a variety of libraries that could be leveraged to streamline the visualization of our crawl statistics. Due to the nature of our tree structure, as well as the complexity of our JSON objects regarding average composites - we found that d3, our initial strategy, was not going to be able to acclimate to the task without intensive research and comprehension on our end. Due to the nature of our project, and maximizing the capabilities of d3 not being the focus or requirement, we leveraged two external libraries, that leveraged the d3 framework, and did the heavy lifting for us in terms of formulating the basis of the visualization. The two aforementioned libraries were, the MUI component library, and the react-d3-tree library. Due to the nature of both libraries, the frontend work required to clean up the statistics data was

much smaller in scope, and the vessels by which the data could be visualized required much less technical knowledge and variance.

7. Testing

In this section, we explore testing to check how well our crawling process and software perform with different user settings. While we are confident in their basic functionality, we want to ensure they work efficiently under various conditions. This testing phase is essential for refining our program, making sure it meets different user needs and operates reliably in real-world situations.

Single URL Text File Tests

Test 1 - User threshold = 0, Paragraph threshold = 0, Hardcount = 10

Finished URL parse in 0.002 seconds

Finished Web Scrape in 2.9008 seconds

Finished and Built Hierarchy in 2.9008 seconds

Test 2 - User threshold = 0, Paragraph threshold = 0, Hardcount = 50

Finished URL parse in 0.0018 seconds

Finished Web Scrape in 11.3094 seconds

Finished and Built Hierarchy in 11.3104 seconds

Test 3 - User threshold = 0.5, Paragraph threshold = 0.5, Hardcount = 100

Finished URL parse in 0.001 seconds

Finished Web Scrape in 0.3678 seconds

Finished and Built Hierarchy in 0.3678 seconds

Multiple URL Text File Tests

Test 1 - User threshold = 0, Paragraph threshold = 0, Hardcount = 50

Finished URL parse in 0.002 seconds

Finished Web Scrape in 12.6762 seconds

Finished and Built Hierarchy in 12.6771 seconds

Test 2 - User threshold = 0.5, Paragraph threshold = 0.5, Hardcount = 50

Finished URL parse in 0.0001 seconds

Finished Web Scrape in 1.2417 seconds

Finished and Built Hierarchy in 1.2417 seconds

Test 3 - User threshold = 0.5, Paragraph threshold = 0.5, Hardcount = 1000

Finished URL parse in 0.003 seconds

Finished Web Scrape in 4.6205 seconds

Finished and Built Hierarchy in 4.6205 seconds

Test 4 - User threshold = 0.7, Paragraph threshold = 0.7, Hardcount = 10000

Finished URL parse in 0.008 seconds

Finished Web Scrape in 16.2923 seconds

Finished and Built Hierarchy in 16.2933 seconds

Conclusions

Starting with single URL crawls, the impact of increasing user and paragraph thresholds on efficiency becomes evident. Higher thresholds streamline the process by eliminating irrelevant webpages. Moving on to text files with multiple URLs, the overall time remains consistent when thresholds are not considered. Notably, a realistic run with high-standard parameters and a substantial hardcount takes approximately 16 seconds. These observations underscore the significance of optimizing user-defined parameters for improved efficiency in handling diverse crawling scenarios.

8. Lessons Learned

Arriving at a consensus for pivotal tasks of the project is crucial to the team aspect of the project. We've learned the importance of communication and time management in order to meet our objectives. Working together and finding middle ground when we face problems is crucial too.

For instance, we all agreed to use Firebase as our database instead of other choices like MongoDB, SQL, and AWS after discussing them.

8.1 Timeline

Below, we have updated the high-level tasks associated with development holistically. This task-by-task timeline was present in our initial presentation, and remains consistent at this point. Any additional tasks previously unconsidered have fallen under one of these previously established umbrellas, which we will cover in their own section.

Project	Status	Notes
9/13 - Preliminary Meetings	Completed	Meet with client to discuss relevant details and constraints with project
9/20 - Mockups	Completed	Create Figma mockups for frontend and create backend structure
10/4 - Frontend Prototype	Completed	Web app is set up and functional with minimal back table end
10/18 - Web Crawler	Completed	Implement web crawler software into web application.
11/3 - Finalize Front End / Visuals	Completed	Final Touches on front end and visuals of descriptive statistics of events
11/28 - Final Testing / Final Crawler	Completed	Testing and bug fixing finished as well as source code

Table 1: Team Development / Planning Increment Timeline

8.2 Problems

In any sizable project, encountering problems is par for the course. This section is dedicated to exploring the various challenges that can arise during such endeavors. By addressing these issues

head-on, we aim to provide a clear understanding of potential pitfalls and empower all involved parties with insights to overcome obstacles and achieve project success.

Backend

In terms of the backend, there were a number of issues that needed to be addressed for the successful implementation of the web crawler.

To start, most of us had never used a web crawler before. As such, research was needed in order to find a suitable library that could handle the parsing of websites. Additionally, we needed a framework to begin the actual parsing and tracking of URLs itself. This would be one of the largest problems throughout the entirety of the project because retrieving the aggregate data from the crawls while maintaining statistics for usage in the frontend requires proper maintenance of individual data nodes and variable tracking.

Possibly one of the most challenging parts of this project was designing a method that would build out the tree structure needed for visualization by the frontend. This was due to a number of reasons such as the need to accommodate the URLThing class, the tracking of URLs throughout the entirety of the crawl, updating necessary JSONArrays, etc. The most crucial issue with the way this is designed stems from the fact that every URLThing has a reference to its parent node using a PID rather than a direct reference to the object. It's likely that future implementations would require some way to change the URLThing object to accommodate this, however the amount of overhead required might be extraneous.

Another challenging yet easily resolved issue involved tracking specific statistics for each crawl. Statistics played a crucial role in providing users with essential insights into each crawl. The challenge was to efficiently store and retrieve these statistics in a format that would facilitate easy display on the frontend.

Parts of this project such as the Firebase and Flask implementation were interesting challenges as several iterations and methods were required to accommodate the needs of the client and frontend to perform CRUD operations.

Frontend

The largest issues with the Frontend development effort, largely consolidated in the Statistics visualization strategy. Due to the nature of how the flask server compiled crawl hierarchical data, and what coveted visualization libraries such as d3 and MUI expected as JSON input format, both the frontend and backend had to rework the strategies by which such hierarchies were built out, and cleaned up. This led to an overhaul of a handful of efforts, those being: the collection & construction efforts of the backend, as well as the data cleanup & data visualization efforts of the frontend.

8.3 Solutions

In response to the challenges encountered during the project, the following section outlines strategic solutions devised by both the backend and frontend teams.

Backend

Starting with the backend, after trying multiple python libraries such as Wayback API, Scrapy, and BeautifulSoup, each member voted on moving forward with BeautifulSoup. We found this library to be the most “programmer-friendly” and was the easiest to comprehend. Moreover, the backend team decided to implement a max heap structure and custom class in order to keep track of the crawled domain. Lastly, in order to keep track of crawl specific statistics, the backend team created global variables that would save each necessary stat and then once the crawl finished, would be sent to the frontend in JSON format in order to be displayed to the user.

The final solution to the tree hierarchy problem relies heavily on the fact that items are placed into the final array in the order that they are parsed through. This means that the parent node is always appended to the array prior to any of its children at some point during the crawl. Abusing this fact, our implementation reverses the array and starts from the bottom before working its way up to build out the tree.

Resolving the statistics issue wasn't overly challenging thanks to effective communication between the frontend and backend teams. Initially, we established a global dictionary to track each statistic, simplifying labeling and storage for the backend. Through clear communication, the backend team crafted a toJSON method to format and send all statistics in a way requested by the frontend team. This streamlined the parsing and display of crawl-derived statistics on the frontend.

Understanding the Firebase and Flask implementation required lots of reading and research to understand the material so that we could put everything together the way we needed it. Additionally, communication and collaboration with the frontend team greatly improved the efficiency of the overall product.

Frontend

Branching off of this streamlined effort to reformat the method by which the backend constructed the statistics object, the frontend team abandoned the direct implementation of a tree structure, and bar chart leveraging base d3. Instead, due to the nature of the project, the complicated nature of the data structure, and the effort required to acclimate to the variance of base d3, the frontend team leveraged two libraries tailor-made for the visualizations required. Those being, the MUI library for bar charts, and the react-d3-tree library for the TreeVisualization. For specific details on each of these libraries, the cleanup & insertion strategy involved, and code excerpts accompanying their key components: we recommend navigating to sections **5.7 & 5.8** and/or the developer manual. Regardless, the implementation of these two libraries enabled us to insert the backend statistics structure into each visualization component, with minimal data cleaning required. This, in turn, streamlined the entire approach to dynamic pagination & statistics visualization, and enabled us to turn in a product more user-friendly and robust from a UI/UX perspective.

8.4 Future Work

There are still ideas we are working on developing and possibly implementing. This section outlines avenues for future exploration, improvement, and innovation.

8.4.1 Linux Server Hosting

Before the semester ends we plan on hosting our project on a Virginia Tech given linux server. This strategic move not only ensures stability and reliability for our work but also enhances security measures, safeguarding the integrity of our project data. Importantly, this hosting choice facilitates convenient access for our client, providing a seamless and secure environment for them to engage with and utilize our project outcomes.

8.4.2 Multithreading

Also, making our web application multithreaded is important. With the user authentication already implemented, making our application multithreaded significantly enhances its performance and responsiveness. This means that multiple users can interact with the application simultaneously, improving the overall user experience by reducing wait times and ensuring efficient handling of concurrent tasks. The implementation of multithreading not only aligns with modern web development standards but also underscores our commitment to delivering a smooth and scalable user interface for our audience.

8.4.3 More Statistics

Expanding our array of statistics holds significance as it offers users a more detailed insight into each crawl. The specific nature of these additional statistics will be guided by valuable input from our client, ensuring that we tailor the information to meet their needs. Additionally, as a team, we will consider other relevant metrics to enhance the overall understanding of our project, providing users with comprehensive and meaningful data.

8.4.4 Greater User Customization

Implementing increased user customization features will enhance the overall user experience. By providing users with more options to tailor their interactions, we aim to create a platform that aligns closely with their preferences and needs. This customization, guided by valuable feedback from our client and team considerations, holds the key to not only meeting but exceeding user expectations, making our project more adaptable and user-friendly.

9. Acknowledgements

Our client - Dr. Mohamed Farag (mmagdy@vt.edu)

We would like to express our thanks to our client, Dr. Farag, for giving us the opportunity to work on this project as well as meeting with us weekly to give us guidance and resources to help accomplish our objective.

10. References

[1] https://react-dropzone.js.org/

[2] https://reactrouter.com/en/main

[3] https://firebase.google.com/docs

[4] https://flask.palletsprojects.com/en/3.0.x/

[5] https://www.npmjs.com/package/react-d3-tree

[6] https://mui.com/x/react-charts/bars/

Automated Crisis Events Collection Builder

USER MANUAL

Team 8

Brian Hays, Parsa Nikpour

CS 4624

October 10, 2023

Table of Contents

1. Introduction

2. User Sign-in

3. Home Page

3.1. User Validation

3.2. Submitting a Crawl

3.3. Viewing Crawls

3.4. Viewing More Crawl Information

4. **Crawl Pagination**

4.1. Understanding the Data

4.2. Navigating Back to Home

Introduction

From the perspective of strictly a user-tailored manual, we will be providing an in-depth overview of the application's frontend, which the user will be engaging with each time they want to utilize the application for its data-side functionality.

IMPORTANT NOTE: *We are assuming that the user currently engaging with this manual understands in some capacity the purpose of this application, what type of input/output is expected as both user-given and system-resulting, and will be able to follow the process outlined below, as they have some level of understanding of the top-level process required for the minimum viable product.*

User Sign-In / User Authentication Page

When a user, who has not previously been cached/saved as a registered user within the system, first reaches the host for the application, they will be greeted with a Firebase mandated "user sign-on" screen. Users who have not engaged with this type of interface before might be lost on where to navigate, so we will provide a step-by-step guide.

I. Provide the authentication form with the information you would like to use as a valid account.
    A. Ensure this information exists, as Firebase will not create a new email address for you.
II. Submit the form to Firebase, and wait for authorization to take effect. You should be redirected to the main home page, in which case, you can proceed to the next step in the manual.

Home Page

Below, we will present an outline for users who have previously leveraged the authentication, or newly leveraged the authentication Firebase provides, and are capable of reaching the application's home page.

I. User Validation

Upon reaching the home page, with the precondition that you are a valid user within Firebase's authentication system, you should be able to look to the top right of the home page's landing banner component, and see a name that reflects the authentication details you originally provided to the form on the login interface.

II. Submitting a Crawl

Once user validation has been completed, a user may wish to submit a crawl. Navigating to this section can be done by scrolling to the next section of the page, which will take you to the interface for uploading a file.

For submitting a crawl, once a user has reached the interface, one can refer to the labeled **figure below** for the process. For clarification, we will explain it in addition. The interface is made up of **two** components. Primarily, you have an upload button, with a validating line of text. This line of text will provide visual feedback to the user, by listing the name of the file they chose to provide the frontend with. **Once a file has been uploaded**, you will have a "set constraints" button present on the interface. This button will only appear when a file has been selected, so as to limit user error. Once this button is clicked by the user, an interface will appear to adjust crawl constraints. One can refer to the **figure below** for a visualization of said interface. For clarification, we will also explain its functionality. A user must provision both a "**crawl name**" and a text file with comma delimited **keywords** for the "predict" algorithm, shown by the mandatory asterisks next to each label. In addition, a user may change the **maximum URLs** checked, the **minimum URL Score** for appending, the **minimum Page Score** for appending, and the **maximum** number of **pages downloaded**. After provisioning at minimum a crawl name, and a keyword file, a "run crawl" button will be present on the interface. This button will only appear when a file has been selected, and the two aforementioned key inputs are filled in, so as to limit user error. Once this button is clicked by the user, one of two things will happen.

I. The crawl was of a successful format for the backend handler, and a crawl will begin, which will send the frontend into an asynchronous waiting state.

II. The crawl was not of a successful format for the backend handler, and a warning message will be thrown back of the form: **“Please insert a text document with valid URLs, each separated by a new line”.**

If a crawl has been successfully submitted, you may navigate to the next section of the manual, which will identify the steps by which a user will take to view their crawl result.

III. Viewing Crawls

If a user wishes to view the crawl they have just initiated, or, **crawls previously generated under this same authentication session**, they can scroll past the submission interface, to find a dynamically generated list of cards.

Each card in this section of the home page corresponds to a crawl previously run by the sessioned user. The cards are simple in information, and contain only the crawl name, the crawl’s URLs that the user provided (the seed URLS), the crawl date, and further user validation by presenting the name of the user who ran the crawl.

If a user wishes to see more information regarding a crawl, specifically the results of that given crawl, please navigate to **Section IV**, which will explain viewing more information for a crawl.

If a user wishes to delete a crawl, they can simply engage with the “delete” UI action on the card itself. There is no user error prevention due to the explicit nature of the text “delete”. Due to this, simply clicking the delete button will fire the necessary API request for deletion, and the card will be removed without further user action.

IV. Viewing More Crawl Information

To view more crawl information, a user simply needs to click on the card for which they want to view more information. Note, that taking this action redirects a user to a page that is tailored to that specific crawl. Backtracking is possible, which will be explained in the next section, but do not take this action unless you are willing to move to a new page, and view new content.

Crawl Pagination

I. Understanding the Data

Once a user has navigated to the relevant page for a given crawl, they will be presented with a compilation of **d3 generated** graphics & statistics. All of these correspond to the crawl a user has previously run. The graphic denoted "Average Score Per Domain" refers to the collection of URLs (both seed and generated) that fall under a specific domain name, and the resulting average score across all URLs from that domain.

Below this graphic, a user will find a Tree Visualization structure. The nodes are color coded, where the **red node** corresponds to the parent for the crawl, being the dummy root node inserted to create a tree structure. **Green nodes** represent leaf nodes without further children. **Blue nodes** represent child nodes that have further children, making them both children of the root (or other parent nodes), and parent nodes themselves. Upon hovering over a blue node, the UI will enhance the scale of that node, and the cursor type will change, providing visual feedback to the user - that clicking that node is a possibility. In addition to this, users are freely able to drag the tree, collapse & reassemble nodes, and zoom in and out - all within the box the tree is contained within. Below, you will find a **figure** showing the tree in view.

If a user has finished viewing data relevant to this crawl, they are free to return to the home page at any time. Please refer to the next section on how.

II. Navigating Back to Home

Once a user is satisfied with any actions taken on the crawl page, they will be able to navigate back to the home page for any purposes they may need. When a user wishes to take this action, they simply need to refer to the top left of the crawl page, where they will see a button with the text **Return to Home** accompanied by a back arrow. If the user chooses to take this action, they will be redirected back to the home page.

Once a user has returned to the home page, they are free to take any of the aforementioned actions, on any of the aforementioned crawls. They are also free to run a new crawl, or

sign out. Please reference each section of the user manual for further instructions on any of these items.

Automated Crisis Events Collection Builder

DEVELOPER MANUAL

Team 8

CS 4624

October 10, 2023

Table of Contents

1. **Dependencies and Installation Instructions**
2. **Frontend**

    2.1 Data Structure Terminology

    2.2 Component Architecture

    2.3 Homepage Overview

    2.4 Server-side Communication

    2.5 MUI and react-d3-tree Visualization & Pagination

3. **Backend**

    3.1 Web Crawler

    3.2 File Parsers

    3.3 URLThing Class

    3.4 Predict

    3.5 Max Heap / Priority Queue

    3.6 Statistics Collection

    3.7 Page Download Storage

    3.8 Firebase Realtime Database

    3.9 Build Hierarchy

1. Dependencies and Installation Instructions

In order to work with the deployment, there are pivotal dependencies involved in both React, specifically the npm libraries & package.json, as well as python. If one wants to work with the code on their own machine, they need to leverage the Docker image files provisioned in both the ./client-final and ./flask-final directories.

Due to the nature of how the frontend interface was developed, the Node.js version involved in the npm library installations, and overall app creation is an earlier version than what most devices run. This being the case, incompatible devices must use the Dockerfile to build, install, host, and work with the image of the frontend despite local device versions of Node.js or absence thereof.

In addition, due to the complexity of the flask server, and the variability of the required python packages / libraries leveraged in our solution, a Dockerfile exists to successfully allow all users, regardless of their native python or libraries, to install and host the flask server in a Docker image. This Dockerfile will leverage the "requirements.txt" file, which can be found in the flask-final directory of the project's overall folder hierarchy.

For explicit details on how to build, run, and engage with each Dockerfile - **you must read the README.md file in the root directory of the source code.** We will omit the lines for building here, but they will be concisely described in the README.md file, as well as where to navigate to run them. A Docker file exists for both the frontend, and the backend frameworks. This is so we can conflate both to their respective ports, and use the route of the flask server for our frontend API calls.

All dependencies will be remediated through the docker images, as they install the dependencies using the required node and python versions for the implementation.

2. Frontend Developer Overview

2.1. Data Structure Terminology

Throughout the following sections, we will make references to the "passed data structure" from the backend. Due to the changing nature of what data this refers to, we have provisioned this section to describe the flow of data over time, and what the structure of the data will look like at each successive iteration and or, within each relevant component file.

CrawlInitializer: CrawlInitializer serves as the entry point for all data. We store our data in firebase in the following fashion: each entry is a unique crawl, with a set of data corresponding to that crawl. Those being, a unique identifier, a set of URLs, a generated tree, statistics accompanying that set of URLs, a collection time, a crawl name, and a submitting user.

When a user signs into the system, the frontend makes an instantaneous request for all crawls that have a "submitting user" value identical to the currently authenticated user. In such a fashion, the returned object becomes an array, where each entry is of the aforementioned structure. This being the case, we then compile this array into a state called **data** which we pass down to **CrawlCardHolder.**

CrawlCardHolder: CrawlCardHolder serves as a mediator between CrawlInitializer, the retriever of data, and CrawlCard, the displayer of data. With this being the case, CrawlCardHolder catches the **data** state as a **react prop** from its parent, CrawlInitializer. Barring any necessary filtering (which we will discuss later) this component then takes the data array, and maps each unique entry (as you recall, each of these is a JSON object) into a CrawlCard component, where the CrawlCard takes each unique element of the JSON object (as described in the CrawlInitializer section above) as a **unique prop**.

With this being the case, each CrawlCard that is generated, has its own unique set of data, that corresponds uniquely and unambiguously to whichever entry in the array generated it. This means that for "n" existing entries in the database for a user, we now have "n"

existing cards in the visual user interface for that same user. Each card is now a vessel by which we store **one entry** of the overarching data array we retrieved initially.

CrawlCard: As mentioned above, each CrawlCard is its own separate instance, containing its own separate entry data. This is leveraged to accomplish pagination. In order to present holistic statistics to a given user, we must first provide them with enough space to hold such information. With this being the requirement, each CrawlCard passes its own unique set of data to the **Crawl Page**, this is done through the ReactRouter library's **Link** component. This task is accomplished in two ways, one - is by using the unique identifier in the object entry as a pagination identifier **[see section 5.4.3 above].** The second, is sending the rest of the statistics objects (namely stats, and tree for the visualization) over to that same paginated route with the **state** field of the **Link** component.

In addition, something worth mentioning - is that CrawlCard has access to a unique property, called **a callback method**. Because the API request to get all user crawls exists in a parent component of CrawlCard, our deletion function must request that refresh from the parent, by calling a function that is scoped to that parent. This is done, similar to how we passed **data** from CrawlInitializer downwards, by passing **an actual function** as a prop down to the child element that requires it. Once this is accomplished, it will refresh **data** at the top-level, and all of the mappings explained thus far will refresh live, and present the updated data array where applicable.

Crawl: Continuing from where we left off in CrawlCard, if a user clicks a CrawlCard in the visual UI, they traverse to the page that contains the mapping mentioned above. That being, a **Crawl Page** with the dynamic parameters filled in by the array entries data. Namely, the "nth" tree and "nth" stats object, that corresponds to the "nth" entry in the data array that the previously clicked card corresponds to. Through this mechanism, users can now see dynamic visualizations of their crawl. **[see section 2.5 below]**

2.2. Component Architecture

IMPORTANT NOTE: *Throughout the explanation in this section, the developers on the project assume that the developer handling this portion of the project has some handle on*

Javascript Frameworks, the way in which they tend to implement web application architecture, and the types of files/styling guidelines/communication best practices that usually accompany such applications.

As things stand in development, the Component Based Architecture for our Crisis Events Collection can be boiled down into three main components. The first is the App.js file. The second level is the two key JSX page files. The third level is all dependent JSX component files, which make up one of the two key JSX page file's and their relevant contents respectively. Within the project file hierarchy, all code pertaining to client side development can be found in the **/client-side** subdirectory.

We will now provide a brief overview on each key file not explicitly visible on the landing page or crawl page, and where relevant content you may be interested in for development purposes will reside.

App.js

The App.js file is equipped with three main functions. The first, is to perpetuate the ./App.css styling down through all its children JSX files. In order to keep styling centralized, but non-confusing, our team leveraged the provided App.js file, and did our own bookkeeping for sectional styling.

On the other hand, App.js has the main function of wrapping all application pages into the **ReactRouter** framework. If you are not familiar, we recommend reading the documentation [https://reactrouter.com/en/main]. The purpose of implementing it in our application is so that we are able to paginate on the basis of "crawl" objects, which we will explain later on.

Home.jsx

The home page is used as a "central hub" to host all of the landing page component files a user will engage with while working with the application. Those being, the landing banner, the **CrawlInitializer**, and the **CrawlCardHolder**. Both of these will be talked about more in the **Homepage Overview**, which can be found next.

Crawl.jsx

The crawl page is used as a "central hub" to host all of the visualization component files / objects a user will engage with while working with the application. Those being, the subcontained **BarChart** MUI component, and child **TreeVisualization** component, which will leverage the **react-d3-tree** component. For more information on this page, and it's components, navigate to **section 2.4**

2.3. Homepage Overview

The homepage serves as the primary landing page and user interface for our application. It is designed with an intuitive layout, ensuring that users with low experience can navigate and operate the application with ease. The page is constructed using various React components.

Key features of the homepage include:

User Authentication Prompt: Ensures only registered users can access the application's functionalities

Crawl Submission Interface/CrawlInitializer.JSX: Allows users to **strictly** input **a .txt file containing** URLs and initiate a web crawl. From a development perspective, we leveraged the **React Dropzone** [https://react-dropzone.js.org/] library, through node packet manager. This JSX file handles the upload and maintenance of **two files** through two useState hooks. Once a user successfully uploads a file, the interface dynamically shows (as shown in the in-line ternary operator in this file) a **set constraints button** allows a user to open an interface. Within that interface, they will be mandated to provision another file, that being one of comma delimited keywords for the "predict" function on the backend. File format validation is not handled by the frontend, but this interface too, leverages the **React dropzone** library. Once this keyword file has been provisioned, alongside **at minimum** a crawl name for the new crawl, and any optional constraints are changed (maximum pages downloaded, and minimum scores), and the user clicks **continue**, a **run crawl** interface button will be made available, which when pressed, sends the file to the backend for processing.

Dynamic Crawl Results Display/CrawlCardHolder.JSX: Displays past and recent crawl results in an organized card layout. Each card provides brief details of the crawl, and users can click on individual cards to view more detailed results. In addition, users are able to filter the queried list of cards, via the input in the jsx. This is done by the onChange parameter of the input fragment. When a change is detected, the event changes the state of the filter to be the new text. Due to the nature of what we map into CrawlCards being **filteredData**, and the fact that the event **setFilter** alters the **filter** state, the function shown in **Figure 1** refreshes **filteredData** as **filter** is used within the function itself, causing a DOM refresh. Apart from filtering, the logistics of this component file are rather simple. We wrap a high-level array mapping function in a div element. For each iteration of the map function, we convert one entry of the data array from the **CrawlInitializer.jsx** file to be a CrawlCard. We pass each array entry unique component objects, as shown in **Figure 2,** as react props to the CrawlCard object. This way, each CrawlCard has access to the component objects unique to its specific entry in the overall data array. **To simplify:** each entry in the data array is transformed into a react object, so that users can engage with them as separate entities. In order for this to work for the Crawl page, we must also pass the Object components (as each entry is a JSON object) to the CrawlCard as react props.

Visual Feedback System: Any action taken on the homepage (like submitting a crawl) provides instant visual feedback, ensuring users are always aware of the system's current status or any potential errors they may face.

```
const filteredData = dataMappable.filter(item => {
  console.log(item["Crawl Name"]);
  return (
    item &&
    item["Crawl Name"] &&
    item["Crawl Name"].toLowerCase().includes(filter.toLowerCase())
  );
});

useEffect(() => {
  setDataMappable(data);
}, [data]);

console.log("dataMappable:", dataMappable);

return (
  <>
    <input
      className="lookup__crawls"
      type="text"
      value={filter}
      onChange={e => setFilter(e.target.value)}
      placeholder="Filter Crawls by Crawl Name"
    />
```

Figure 1: Crawl Card Filtering

```
<div className="card__holder">
  {filteredData.map((crawlData, index) => (
    crawlData.message === "data in correct format" ? (
      <div className="card__section" key={index}>
        <CrawlCard
          id={crawlData["Crawl ID"]}
          name={crawlData["Crawl Name"]}
          crawlName={crawlData["Crawl Name"]}
          user={user}
          urls={crawlData.URLs}
          stats={crawlData.Stats}
          date={crawlData["Collection Time"]}
          tree={crawlData.Tree}
          childFunction={childFunction}
        />
      </div>
    ) : (
      <div key={index}>Not mappable</div>
    )
  ))}
</div>
```

Figure 2: Data->Crawl Card Mapping

2.4. Server-side Communication

Server-side communication refers to the interaction between the frontend (React application) and the backend (server hosting the web crawler and other functions). This

interaction will be facilitated using the **Axios library**, which will allow for easy HTTP requests to the server. Each of the HTTP requests described below follow an identical format, which is accomplished as described next.

Every HTTP request is accomplished via an asynchronous javascript function. You can see the structure in **Figure 3**. First, a constant API url is set to be the accompanying flask route for the python function necessary to accomplish the task. Next, we instantiate a **FormData** object. This is done so that we can append several necessary variables from the frontend, and send them to the backend, so that we can either build out a required object (i.e. run a crawl with a file the user provisions in the below image) or, as you will see in post requests such as the delete method present in **CrawlCard**, we send in the Crawl ID unique to that card, so that the flask server can use it to delete the firebase data entry. After form data is appended with the relevant variables. We await our axios post method, by sending the **form data** to the **api url** and asynchronously catching the response via a promise. Once this data is retrieved, we set the data state using **setData** to be the response from the asynchronous post/get/delete methods. In error, we log errors. Otherwise, we now have the data required, or have accomplished the task necessary, and can execute a function that requires such an HTTP request as a prerequisite.

```
async function sendDeleteRequest(){
  try{
    const apiUrl = 'http://127.0.0.1:5000/delete_firebase_entry'
    formDataDelete.append('crawl_name', crawlName);

    const responseGet = await instance.post(apiUrl, formDataDelete, {
        headers:{
          'Content-Type': 'multipart/form-data'
        },
    })

    childFunction();

  }
  catch(error){
    console.error('Failed to Delete:', error);
  }
}
```

Figure 3: API Request Format

Key points of this section include:

GET Requests: Used to retrieve existing crawl data from the server. This will be leveraged, as previously mentioned, in **CrawlInitializer.jsx**, which will make an axios GET request to the backend's defined flask route for data.

POST Requests: Allow users to send crawl instructions (URLs) to the server. This will be leveraged, as previously mentioned, in **CrawlInitializer.jsx**, which will make an axios POST request to a route within the backend flask server. In addition, **CrawlCard.jsx** contains a POST request to the deletion method in the backend flask server. This is performed when a user engages with the "delete" button. Once the POST request is complete, a parent function passed as a react prop down from **CrawlInitializer.jsx** will re-fire the GET Request mentioned above, acting as a refresh to eliminate the deleted firebase entry from the list of all cards.

Real-time Feedback: As the backend processes crawls, it sends updates to the frontend, allowing for real-time progress updates to the user

Error Handling: Any issues faced by the backend during crawling (like invalid URLs) are communicated to the frontend, which then provides the user with an appropriate error message.

2.5. D3 Visualization / Pagination

This section focuses on how the application will visually represent crawl data and manage the display of large data sets. Implementation will involve introducing new React components for visualizations and integrating the D3.js library. Pagination will involve restructuring how crawl data is fetched and displayed on the homepage,

MUI & Tree Visualization: Leveraging the MUI & react-d3-tree libraries, the application visually represents our statistics two-fold. One being, a composite bar graph that represents the average score per-domain generated by the crawl. The second, being a user-interactive tree visualization, with collapsible & uncollapsible nodes, to explore the process by which each node generated its children. In order to make these visualizations dynamic, the Crawl Page takes in the relevant statistics object & tree object through the

state component of the ReactRouter **Link** component. Due to the nature of this library, and react props in general, because we have dynamically mapped our array of "n" JSON object entries into "n" CrawlCards, Crawl Cards only have access to the data that exists for their specific "nth" entry. This being the case, when the id & state are sent to the "nth" page that this CrawlCard visits **(see Pagination for how this works)** the data the MUI Bar Graph, and Tree Visualization utilize are specifically tied to the crawl card clicked.

Data Cleaning: In order to successfully leverage the aforementioned libraries, there was some data cleaning involved. In order to leverage both the Bar Chart from MUI, and the Tree from react-d3-tree, we needed to reduce each node / each JSON entry, to only have two parameters each.

[The below excerpt can also be found in section 5.7 of the overall document]

For The Bar Chart, due to the nature of the firebase storage, and what is valid in terms of structure, what the Crawl.jsx file has access to, is a JSON object that has been converted & stored in string format. This being the case, Crawl.jsx leverages a two step process, one - being the JSON.parse function, and the other being the Object.keys function. These two functions, in tandem, generate an object structure that the MUI library can correctly leverage to generate bar charts. The code is very explicit, but at a high level, the data component determining the parameters of the bar graph, takes in this newly parsed data, **termed "websites"** and maps it holistically into just its average URL score. This "average score" is coined & computed via the flask server, so the frontend simply needs to use its existence for the bar chart. For a more concise explanation of the structure of the **stats** and now **websites** objects, the following is true: each domain found in the history of the crawl, has a count of the amount of times it appeared, an array of all scores across those appearances, and then a composite average taken over that array. All of this data is computed by the backend, and we simply map each domain entry to only be its average score, utilizing the domain name for the xAxis, by not mapping it whatsoever. Again, the code is rather explicit, but the differentiation between only using **websites** and using **websites mapped to be its avg_score component** is what allows the bar chart to generate its x vs. y relationship visually for all domains retrieved in the crawl.

[The below excerpt can also be found in section 5.8 of the overall document]

For the Tree Visualization, in order to refine the several-component-object entries of our tree structure, into one that only possesses two, those being **name (for displaying text per node)** and **children (for containing all subsequent nodes in the hierarchy, with the given node as it's parent)**. This is accomplished by passing our newly formatted data (with the dummy root prepended) to a function that leverages the currently existing JSON object parameters, and compiles the relevant ones into a "name" parameter that contains the following:

"Domain name, Average Score, Amount of Children, Average Children Score"

This is accomplished by formatting the URL, and retrieving the base URL domain, formatting the pre-existing "Avg_score" parameter to a fixed length, and retrieving the child lengths & average composite scores recursively, based on the pre-existing "children" parameter the passed **tree** object contains. Once this data cleanup is complete, we have a hierarchical, node-based JSON object structure, where each node is a key-value pair of name-children. With this being the case, we simply pass this object to the **react-d3-tree [5]** Tree component. There is some accompanying logic regarding scaling, path curvature, zoom maximization, and CSS classes for visualization - but all of these can be found in more extensive detail through the documentation found by referencing the following: **https://www.npmjs.com/package/react-d3-tree**

Interactive Visuals: As mentioned, users will not just view static charts; they will be able to interact with them. Hovering over a segment of the bar chart, will display the average score. Hovering over color-coded nodes (as described above) will enable users to open nodes that have child URLs in the hierarchy, or collapse said nodes to make the UI cleaner, and explore other nodes.

Pagination: As users run more crawls, the amount of data will increase. Pagination exists to manage the display of this data. Instead of loading all results at once which can be slow, results will be divided into "pages". Users can navigate between these pages, ensuring faster load times and a more organized view. As previously mentioned, this will be carried out by an "onclick" function present on each card's jsx code. This function will

redirect the user to a dynamic route corresponding to that crawl's id. This logic is built out in **two key files,** those being, as shown in **Figures 4** and **5**, **index.js** and **CrawlCard.jsx**. Index.js accomplishes the necessary ReactRouter setup, by which we create two separate routes. One, is the "/" route as shown in the figure, which leverages the <App/> component (which is simply App.js) the other, is the "/crawls/:crawlNumber" component, which leverages the <Crawl/> component, which is the **Crawl.jsx** file and page. The :crawlNumber being appended at the end, means that we need an identifier to reach this route, after the /crawls/ routing. This is accomplished by CrawlCard. As shown in the figure, we use a Link component from ReactRouter, to visit **/crawls/${id}**. Id, is a prop passed from **CrawlCardHolder**. As we discussed in that section, the mapping of our data array retrieved in **CrawlInitializer**, also provisions a ***uniqid*** field into the firebase entry for each crawl, and thus on retrieval, each array entry for the data object. This Id is then sent into each subsequent CrawlCard mapping, as a unique identifier. Due to the nature of props in react, this means each CrawlCard instance has its own unique set of props. This being the case, linking to **/crawls/${id}** actually accomplishes the dynamic pagination requirement, as no two cards will link to the same page. As you can see in the figure as well, there are some additional **state** parameters being sent to the Crawl page. This is done so that in addition to the unique ID, the unique data each CrawlCard entry possesses will exist at that new route, so that the visualizations are able to dynamically show each crawl in a unique fashion. For more information on how, reference the above section on the Visualizations.

```
import {
  createBrowserRouter,
  RouterProvider,
} from "react-router-dom";

const router = createBrowserRouter([
  {
    path: "/",
    element: <App/>,
  },
  {
    path: "/crawls/:crawlNumber",
    element: <Crawl/>,
  }
])

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <RouterProvider router={router}/>
  </React.StrictMode>
);
```

Figure 4: React Router Setup

```
<div className="crawl__card--actual" >
<button className="close__button--card" onClick={sendDeleteRequest}>Delete Crawl</button>
<Link to={`/crawls/${id}`} state={{displaySeeds, stats, tree, id, crawlName, email}}>
  <div id={id} className="crawl__wrapper">
```

Figure 5: Link Functionality via. React Router

3. Backend

This section will cover strictly the items located in backend.py. The items within this section relate to the web crawler, the structure and objects used, the handling of statistics, etc.

3.1 Web Crawler

The web crawler code structure is based on the Python library, ***Beautiful Soup*** which contains various useful web scraping tools necessary for our project. The exact process of the web crawler can be described as follows:

1. The user inputted initial URL list is parsed and separated using both of the relevant file parsers (Section 2.2). With each URL being turned into a **URLThing** object (Section 2.3) and inserted into the list with an Average Score of 1.0.
2. The URL with the highest score is popped from the top of the heap and its contents and metadata are downloaded using BeautifulSoup.
3. The contents are scored, downloaded (Section 2.7), and then parsed through to find other URLs (child URLs).
4. These child URLs are then scored based on the average score of its anchor tag text and parent URL's paragraph text obtained by the predict function.
5. Statistics related to the URL's score, common domain, and placement in the overall crawl's tree are collected and stored into a JSONArray.
6. The child URLs are then inserted back into the heap.
7. Steps 2 through 6 are then repeated until the user-established hard count for number of URLs parsed is hit or there are no more URLs within the heap.

8. The final JSONArray containing all of the statistics and parsed URLs are then pushed to the database for future use.

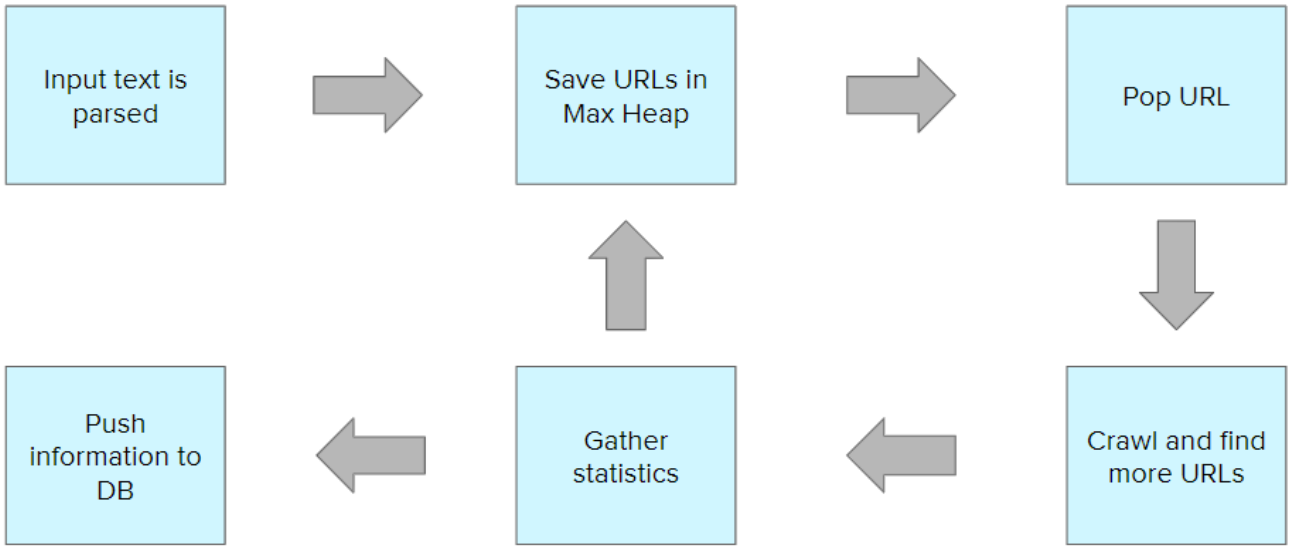

Figure 6: Crawler Diagram

3.2 File Parsers

There are two methods within the backend.py program that handles the reading and parsing of text files; the **read_urls_from_file** and the **read_keywords_from_file** method.

read_urls_from_file: This method is used for the user's inputted list of valid URLs. The method (and by extension the program) assumes that the user is giving us a file containing a list of newline delimited URLs that are all perfect to act as a hub node within the tree. Based on this fact, the method reads the text file it is given, parses line by

line, and creates a new URLThing object out of each URL that is found. The newly created URLThing is then inserted into the max heap, appended to the crawlJSONArray, and counted towards the total number of parsed URLs.

```
# Function to collect URLs from user input and push URLs to heap.
def read_urls_from_file(file_name):

    with open(file_name, 'r') as file:
        # Read the content of the file and split it into individual URLs
        urls = file.read().split('\n')

        for url in urls:
            global count
            hold = URLThing(count,url)
            push_to_heap(hold) # Given URLs will be assumed as valid
            crawlJSONArray.append(hold.toJSON())
            count+=1
```

Figure 7: read_urls_from_file method

read_keywords_from_file: This method takes a file with keywords inside of it and fills up the keywords list variable with the words found inside. The words must be delimited using either a '|', a new line, or a comma. The keywords variable list is then parsed through to remove whitespace before returning the final list.

```
# Function to collect keywords from user input and push URLs to heap.
def read_keywords_from_file(file_name):
    fin = []

    with open(file_name, 'r') as file:
        # Read the content of the file and split it into individual URLs
        hold = re.split(r'[\|\n,]',file.read().strip(' '))

    # Whitespace clearing
    for item in hold:
        fin.append(item.strip())

    keywords = fin
```

Figure 8: read_keywords_from_file method

3.3 URLThing Class

As mentioned above, all URLs that are pulled from the list of user requested URLs are turned into a URLThing object. This class is used to make URL tracking easier and enables a hierarchical representation of the URLs once crawling is completed. There are a total of six defined functions for the URLThing class including the init needed to define the values within it.

Upon URLThing object creation, the following seven fields are populated:

Field	Type	Use	Default Value
ID	Int	Used to track each URL	
URL	String	Used to hold the URL	
URL Score	Float	Used to hold the url_score	1
para_score	Float	Used to store the parent's para_score	1
avg_score	Float	The average value of the url_score and para_score	1
pid	Int	Used to track the parent the URL was pulled from	None
children	List	Used to hold the children of the URLThing	[]

Table 2: List of URLThing Class Fields and Use

This is done using the __init__ constructor defined for the URLThing class.

```
class URLThing:
    def __init__(self,id,url,url_score=1,para_score=1,avg_score=1,pid=None):
        self.id = id
        self.url = url
        self.url_score = url_score
        self.para_score = para_score
        self.avg_score = avg_score
        self.pid = pid
        self.children = []
```

Figure 9: URLThing Class

__lt__: The less than function or "__lt__" is used internally by the URLThing for comparison which is required by the Max Heap in order to correctly sort the items inside. The comparison is based on the average score of both URLThings.

```
# Less than comparison (<) for heap.
    def __lt__(self, other):
        return self.avg_score > other.avg_score
```

Figure 10: __lt__ Comparison for max heap

__str__: This method returns a string containing the URLThing's URL, ID, Average Score, and PID. This method was primarily used for testing purposes.

```
# To String
    def __str__(self):
        return f"(URL: {self.url}, ID: {self.id}, Avg Score: {self.avg_score},
PID: {self.pid})"
```

Figure 11: __str__ method

toJSON: This method converts the fields within the URLThing into a JSONArray which is used for data storage and retrieval. The children field is special and for each child

within its list it performs the toJSON method on it in order to get all of the important information related to each child URL while maintaining the hierarchical structure.

```
    # Method to convert URLThing to JSONArray
    def toJSON(self):
        return {
                "ID": self.id,
                "URL": self.url,
                "URL_Score": self.url_score,
                "Para_Score": self.para_score,
                "Avg_score":self.avg_score,
                "PID": self.pid,
                "children": [child.toJSON() for child in self.children]
            }
```

Figure 12: toJSON method

Add_child and remove_child: These two functions are pretty self explanatory. They are used to append and remove children from the URLThing's children list.

```
# Adds children to children list.
    def add_child(self,child):
        self.children.append(child)

    # Removes unwanted children from the list.
    def remove_child(self,child):
        self.children.remove(child)
```

Figure 13: Methods to add/remove from list

3.4 Predict

The current predict function is a quick implementation that we expect will be switched out in favor of the LLM that several of the other groups are working on. For now, the predict function averages the number of times a keyword is located within the input string.

```
# THE PREDICT FUNCTION
def predict(input_string, keyword_list):
    # Convert the input string to lowercase for case-insensitive matching
    input_string = input_string.lower()

    # Initialize a variable to keep track of the number of keyword matches
    num_matches = 0

    # Iterate through the keyword list and check for matches in the
input_string
    for keyword in keyword_list:
        # Use a regular expression to search for the keyword in the
input_string
        if re.search(r'\b' + re.escape(keyword.lower()) + r'\b',
input_string):
            num_matches += 1

    # Calculate the percentile score as the ratio of matches to total
keywords
    percentile_score = (num_matches / len(keyword_list))

    return percentile_score
```

Figure 14: Predict method

To start, the predict function takes a string and a list of keywords. The input string can be any line of text. However, for the purposes of our crawler it is strictly used to analyze paragraph and anchor text. We expect the keyword list to be a global variable that is populated once the user inserts a keyword list and begins a crawl. Once the accepted inputs are inserted, the words within the keyword list are iterated over sequentially. At each word, the string is checked using regex to see if the keyword is anywhere within the inputted string. If the keyword is found, a counter is increased by 1. Once all of the keywords have been iterated over, the ratio of found keywords to number of keywords is calculated and returned.

3.5 Max Heap / Priority Queue

The max heap or priority queue, is an essential piece of the processing and filtering of URLs. As we parse through each URL we will inevitably come across child URLs that will have varying levels of relevancy. As such, once each URL is scored there needs to be some way to sort the URLs so that the most relevant URLs are always crawled through first.

Our max heap utilizes the heapq python library which simplifies the process greatly. Using this library, we are able to create a list called **prio_heap** and utilize two functions (**push_to_heap** and **pop_heap**) to control the addition and removal of items from the heap. Both of these methods do as implied.

push_to_heap: This method takes a URLThing, and with the help of the heapq library sorts it into the **prio_heap** using the URLThing's builtin less than comparison (Section 2.2).

pop_heap: This method removes the first URLThing off the top of its heap and returns it.

```
# Priority Queue
prio_heap = []

def push_to_heap(urlThing):
    heapq.heappush(prio_heap, urlThing)

def pop_heap():
    return heapq.heappop(prio_heap)
```

Figure 15: Pop/Push methods for max heap

3.6 Statistics Collection

One of the key aspects of our project is the collecting, storing, and displaying the data of user's crawls in an effective and useful manner. In order to do this, we employ an in-line collection of various statistics such as:

- Domain of URL.
- Counts of domains of anchor tags that pass threshold values.
- An average of two scores (para_score and anchor_score).
- Count of total URLs.

All of these statistics are stored in an object called crawlJSONArray (with the exception of the counts of domains which already have their own object). This array holds these URLs as strings that are in the format of a JSONObject for easy retrieval and storage in our database.

The other data object that stores our statistics is a dictionary named statCrawlSucc that contains a key value of a domain of a URL. The value part of the pairing represents the total number of URLs that passed the threshold and share the same domain.

3.7 Page Download Storage

During crawling, web pages are saved as HTML files in Firebase storage using the **pop_scrape()** method. Each crawl is assigned a unique crawlID, which serves as a folder to store the HTMLs. Additionally, every downloaded HTML is given its own ID, starting from 0. For instance, if the crawlID is 2 and the HTML ID is 1, the file is stored in Firebase storage as "2/1.html."

```
file_name = f'{url.id}.html'
# Specify the path to the HTML file within the folder
file_path = f'{crawlId}/{url.id}.html'
try:
     blob =
storage.bucket(app=firebase_admin.get_app(),name='auto-scrape-crisis.appspot.co
m').blob(file_path)
     bucket = storage.bucket()
     blob = bucket.blob(file_path)

     blob.upload_from_string(response.content)
     print(f'Successfully uploaded {file_name}')
except Exception as e:
     print(f'Error: {e}')
```

Figure 16: Downloading htmls to Firebase Storage code

To retrieve all HTMLs from a particular crawl, we employ the **download_from_storage(id)** method. This function requires the user to provide the unique crawlID they want to fetch HTMLs from. Using this ID, we search the Firebase storage for a corresponding folder. Once located, we iterate through the folder, downloading and appending each HTML to an array. Finally, the array is returned to the user.

```
def download_from_storage(id):
    arr = []
    bucket = storage.bucket()
    # All folders will be 'crawlId'/
    folder_path = f'{id}/'
    # List all objects in the specified folder
    blobs = bucket.list_blobs(prefix=folder_path)

    for blob in blobs:
        local_file_path = blob.name[len(folder_path):]
        # Download the file
        if local_file_path != '':
            try:
                arr.append(local_file_path)
                print(f"File downloaded: {local_file_path}")
```

```
            except Exception as e:
                print(f'Error downloading file: {local_file_path}')
    return arr
```

Figure 17: download_from_storage method

3.8 Firebase Realtime Database

In order to store our data we will be using the cloud-hosted Firebase's realtime database. Every entry into the database will be a JSONArray containing information from each crawl. This crawl will contain all URLs that were deemed as valid as well as statistics such as number of common domains, a tree structure of the order in which the webpages were crawled, and a final weighted score based on the average scores derived from the children URLs. This is done using the **run_to_db()** method.

run_to_db: This method works by formatting the global variables that track statistics such as the crawlJSONArray, crawlStatSucc, tree array, and hubs into one JSONArray. This array is then pushed to the Firebase DB using the necessary database reference and credentials.

```
def run_to_db():
    global db_item_count, hubs, crawlJSONArray, crawlStatSucc
    # Get the current date and time
    collection_time = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')

    if db_item_count is None:
        db_item_count = 0
    temp = build_hierarchy(crawlJSONArray)
    # Save the data to Firebase
    data = {
        "Crawl ID":db_item_count,
        "Collection Time":collection_time,
        "Name": "Trevor",
        "Crawl Name": "Monterey Park Shooting",
        "URLs": crawlJSONArray,
```

```
        "Number of Hubs": hubs,
        "Stats": json.dumps(crawlStatSucc),
        "Tree": temp
        }

    db_ref.push(data)
```

Figure 18: run_to_db method

3.9 Build Hierarchy

The build_hierarchy method takes the crawlJSONArray and converts it into a nested array structure which allows for ease of visualization by the frontend. In order to achieve this, the function takes advantage of the fact that items are inserted into the crawlJSONArray in the order they are parsed. This means that the parent URL will always be inserted into the array before its children, saving us the time of sorting the items in the array. The function traverses through the dataset in reverse order and builds a dictionary that holds the IDs of the children with their respective PIDs as a key. If a PID that is in the dictionary is located, the children are appended to the children field of the parent URLThing. The children URLs are then deleted from the dictionary and removed from the dataset to prevent duplicates. Once the entire dataset is parsed through, the function returns the nested array.

```
# Function that converts the JSONArray to a hierarchical tree structure.
def build_hierarchy(data):
    global hubs

    # Create a dictionary to temporarily store items that couldn't be added
initially
    parent_to_child = {}

    for item in reversed(data):
        id = item['ID']
```

```
        pid = item['PID']

        if id in parent_to_child:
            item['children'] = parent_to_child[id]
            del parent_to_child[id]

        if pid not in parent_to_child:
            parent_to_child[pid] = [item]
            hubs += 1
        else:
            parent_to_child[pid].append(item)

    return parent_to_child
```

Figure 19: build_hierarchy() method