# CS4624: Crisis Events Knowledge Graph Generation

Developers: James De Chutkowski, Justin Turkiewicz,
Geethika Abhilash, Anthony Tran,
Matthew Walters

Client: Dr. Mohamed Farag
Virginia Tech
Blacksburg, Virginia

May 1, 2024

**Abstract**

In a world inundated by information during crisis events, the challenge isn't just finding data, it's making sense of it. Knowledge graphs rise to this challenge by structuring disparate data into interconnected insights, enabling a clearer understanding of complex situations through visual relationships and contextual analysis. This report presents a web-based application for generating and managing knowledge graphs and details the process taken to create it. The application integrates React with Material-UI for the frontend, Flask for the backend, and MongoDB and Neo4j for data storage. Users input multi document collections which are processed using Beautiful Soup, Stanford's Core NLP, NLTK and SpaCy to extract and analyze data, forming triplestores and named entities. These elements are then used to generate knowledge graphs that are stored in Neo4j and rendered on the web via Sigma.js and Graphology. This report addresses development processes, features, testing, step-by-step guidance for users and developers, the lessons we learned working on this project, and potential enhancements that can be implemented by future student groups picking up our project.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

In the face of crisis events such as natural disasters or pandemics, the ability to process information and coordinate responses is crucial. Knowledge graphs, which illustrate the complex interrelations between entities in a given domain, emerge as a vital tool in structuring and understanding the intricate web of crisis-related data. Through a user-centered design approach, coupled with graph database technologies, we have created a platform that facilitates the exploration of data. This paper details the development of a user interface (UI) designed to enable stakeholders to effectively generate, visualize, and interact with knowledge graphs tailored to crisis events. The subsequent sections of this report outline the team organization, detailed requirements, distinctive features of the solution, and the technical implementation, including frontend, backend, and database considerations. Additionally, user and developer manuals provide guidance on system operation and maintenance, while discussions on data files, deliverables, and team roles give insight into the project's execution. The document concludes with reflections on the challenges faced, lessons learned, and a vision for the expansion and enhancement of the system we created this semester for future groups to build upon in the coming semesters.

# 2 Team Organization

**James de Chutkowski**
Team leader and Frontend developer. In charge of home page screen and user interface. Working on integration between frontend and backend. Worked on coordination with client.

**Matthew Walters**
ML and Page Scraping developer. In charge of scraping the URLs and extracting text from HTML files. Also working on extracting the triplets from the text with an ML model.

**Geethika Abhilash**
Frontend developer. In charge of rendering and laying out the knowledge graph to the user. Also working on integrating the frontend and backend.

**Anthony Tran**
Frontend and backend developer. Worked on creating the login and sign up pages along with the user info page. Created backend API endpoints to interact with user account info. Setup MongoDB database to store user account info.

**Justin Turkiewicz**
Backend and DevOps developer. Worked on creating the knowledge graph endpoints to interact with the graph information. Setup Neo4j database to store graph info. Will work on containerizing application.

# 3 Motivation

In times of crisis, finding comprehending information can be overwhelming due to the vast amount of data generated and the high stakes involved in decision-making for these kinds of issues. Stakeholders, such as emergency responders, public health officials, and especially the public, often face challenges in navigating through noise and unstructured data. The criticality of graphical information representation becomes paramount in understanding the impact of the crisis, coordinating the appropriate response efforts, and ensuring the safety and well-being of individuals and communities. Consequently, there is a pressing need for systems that can efficiently organize, present, and analyze data, making it easily accessible and interpretable. This requirement underlines the importance of developing advanced tools like knowledge graphs integrated with user-friendly interfaces. These technologies can synthesize complex information from diverse sources into coherent insights, thereby empowering users to make informed decisions in the aftermath of crisis events.

# 4 Requirements

Users are able to generate and view knowledge graphs from the information extracted from multiple documents. The user must be able to do this via an interactive web UI. Users should have their own account login, which is how they manage and view their collections and their corresponding knowledge graphs in the UI. There should be an admin user which can view other users collections and manage them.

## 4.1 Collections

Users can create multiple different collections of documents. Selecting a collection displays a knowledge graph associated with the collection's documents. The user can delete collections and add new collections. Users can only view collections made by them, and admin users can view collections of all users. Modifying existing collections is not supported for this project.

## 4.2 User Input

The user is able to create a new collection with one of the three following input types.

- Single text file with URLs
- Zip file of HTML documents
- Zip file of text documents

All input types are currently mutually exclusive. A user cannot submit a text file and a zip file, and one zip file cannot contain different types of documents.

## 4.3 Multiple Users

Functionality for multiple users is supported through a login system. Each user will have their own set of collections and graphs saved. An admin user has their own account similar to other users but also has privileged abilities above other users.

## 4.4 Data Storage

User login information is stored in a MongoDB database. Text extracted from inputted documents is saved to a local database. Each collection has an associated graph stored in a graph database.

# 5 Project Design & Architecture

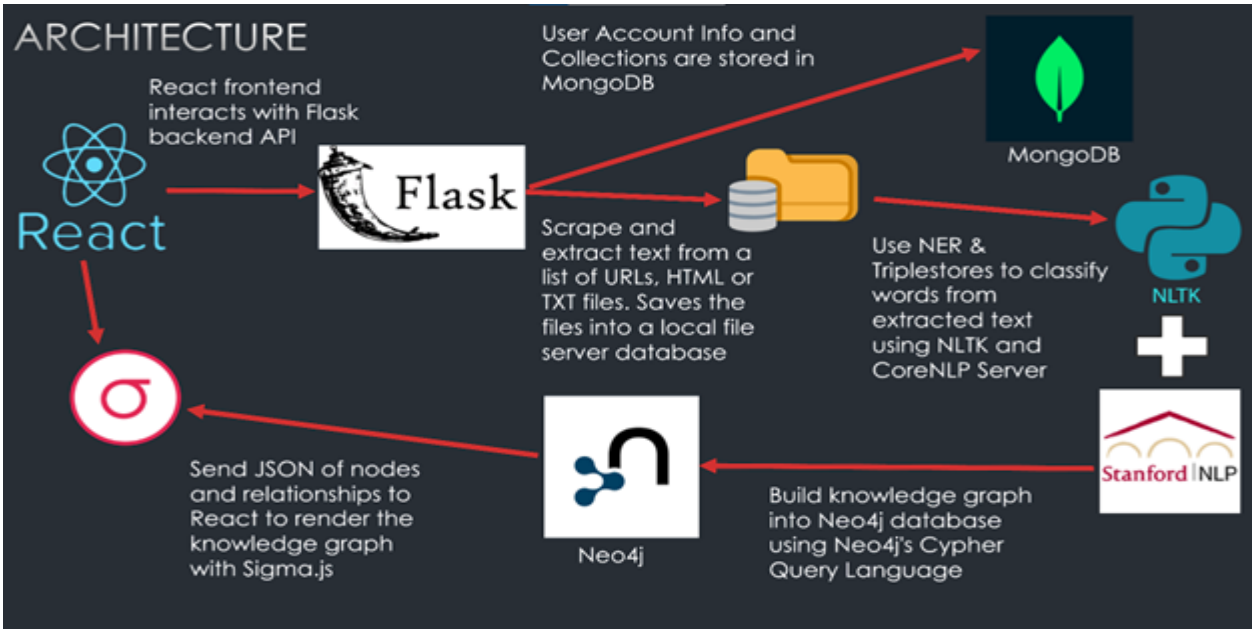


Figure 1: Diagram of the projects architecture

The project's architecture is broken into 6 main components:

1. React website
2. Flask backend
3. Stanford CoreNLP processor
4. Neo4j database
5. Mongo database
6. File Server database

The react website is the client facing frontend of the application. Users interact with the application through the website. The react server makes API calls to the flask backend based on the users actions on the website. When a user gets their current collections on the sidemenu, the flask backend gets the list of the users current collections stored in the file server database. This can be done through the Mongo database as well since their is a collection called "collections" that stores this information as well. The file server database was chosen as it is quicker to access than the Mongo database. When a user selects a collection, the flask backend will query the nodes and relationships for that particular collection from the Neo4j database and return that to the react client. The react client then renders the graph with Sigma.js. When a user wishes to view the collection info ("metadata" field from the GET /api/knowledgegraph request), the various fields for the collection info is retrieved from all 3 databases. The node and relationship count is retrieved from the Neo4j database, the file name(s) and document count is retrieved from the File server database and the time the collection was created is retrieved from the Mongo database. When a user deletes a collection, the information about the collection is deleted from all 3 database services. Similarly, when a user creates a collection, the information about the collection is stored in all 3 database services. To get the subject-predicate-object triples from the user inputted text, the application uses Stanford's CoreNLP ML model. This model is run locally as a server. The flask backend posts the inputted text to the CoreNLP server and

the server responds with the subject-predicate-object triples it formed from the inputted text to the flask backend server. The flask backend server then uses these triples to create the nodes and relationships of the graph in Neo4j.

# 6 Features

The application contains various functionalities that can be categorized into two groups: user-oriented features and graph-related features.

## 6.1 User-Oriented Features

- User Authentication System: Our application lets users access their accounts. This system verifies user credentials to allow only registered users to utilize the application.
- User Account Creation: Our application also allows users to make their unique accounts. This feature allows new users to create accounts with their relevant information during registration.
- User Logout Functionality: Our application incorporates a user logout feature, granting users control over their sessions. This functionality enables users to sign out when they finish their sessions.

## 6.2 Graph-Related Features

- Collection Selection: The user can choose different collections to view the associated knowledge graphs by selecting them from a side menu. The collection item will also display which user owns the collection.
- Graph Viewing and Interaction: Users can visualize and explore graph data using interactive tools. This feature allows users to gain insights, analyze relationships, and make informed decisions based on graph representations within the application.
- Creating Collections: At the top of the side menu is a "Create New Collection" button that opens a modal prompting the user to insert the name of the new collection and a file selector asking them to input a file to be sent to the backend to be used to generate a knowledge graph.
- Deleting Collections: Our application includes a feature enabling users to delete their created collections. This functionality allows for the removal of collections no longer needed.
- Collection Information Viewing: Users can view detailed information about their collections such as what documents are included to gain insights into the content and structure of collections used for graph generation.

## 6.3 Admin Features

- Admin users have all the abilities normal users have
- Admin users can also delete collections from other users.

# 7 Implementation

Our application is structured into distinct components, comprising the frontend, backend, database, and APIs. The frontend facilitates user interaction with our application. It handles user inputs and renders graphs. Meanwhile, the backend manages background tasks, connects to the database, and handles parsing logic. Our database is the repository for storing user and collection information, as well as graph data. Additionally, our APIs serve as the communication interface between the frontend and backend systems.

## 7.1 Frontend

In our frontend development, we leverage React [5], a popular JavaScript library for building user interfaces, along with libraries like Material UI [8] to enhance the aesthetic appeal of our user interface. We adhere to common frontend design patterns, which include utilizing React Router for efficient page routing and adopting a component-driven development approach.

### 7.1.1 Login Page

For our login functionality, we've implemented a interface using Material UI's text fields and buttons. Upon pressing the login button, an Axios call is triggered, connecting to our backend's login endpoint. This endpoint processes the user's credentials and responds with a status indicating the success or failure of the login attempt, accompanied by an error message if applicable. If a user opts to create a new account, they can do so by clicking the "Create New Account" button. This action opens a Material UI dialog, allowing the user to input their information. Upon submitting the form, an Axios call is made to our backend's account creation endpoint. Similar to the login process, the endpoint returns a status indicating the success or failure of the account creation, along with any relevant error messages.

### 7.1.2 Home Page

The main page was developed using React.js, along with Material-UI (MUI) components to ensure a clean and simple design. The home page serves as the main interface, composed of various embedded React components seamlessly integrated. Embedded components include a container for displaying knowledge graphs and a side menu for holding collections. We employed modals extensively throughout the interface, which are pop-up dialog boxes used for displaying information or capturing user input without navigating away from the current page. These modals were implemented for functionalities such as creating and deleting collections, accessing collection information, and managing user details. The combination of React and MUI facilitated the development of a dynamic and responsive frontend, enabling smooth navigation and efficient interaction.

### 7.1.3 Graph Rendering

When the user selects a collection, our frontend requests a graph through the flask API. The graph is retrieved from the Neo4j graph database and sent to the frontend in the form of a JSON file for rendering. The nodes, edges, and relevant labels are extracted from the JSON and plotted using a combination of Sigma and Graphology[4]. Colors of the nodes are randomized. Node placement is determined using Graphology's forceAtlas2 algorithm. This causes nodes to repel each other except when joined by edges which cause nodes to attract.

## 7.2 Backend

In our backend infrastructure, we use Flask, a Python framework that facilitates client connections to our database, enabling data querying and additional functionalities. We utilize MongoDB Atlas, a virtual hosting service, to store user and collection-related information securely. For managing graph data, we utilize Neo4j as our graph database, while Sigma is used to parse graph data into a user-friendly format for visualization on the frontend.

### 7.2.1 User Authentication

In our user authentication backend, we have implemented two primary pathways: logging in a user and creating a user account.
For logging in a user, the process begins by parsing the username and password data received from the POST call. We then validate if the username exists in our database. We then compare the encrypted password stored in our database with the provided password using bcrypt's [7] checkpw function. If the credentials match, we return a success status along with relevant data such as the user's username and permissions type through a cookie. Conversely, if any of these verification steps fail, we return a fail status accompanied by

an error message detailing the reason for the failure.
On the other hand, for creating a new user account, we first verify if the username provided in the POST call already exists in our database. If it doesn't, we proceed to hash the user's password using bcrypt. Following this, we insert the user's name, email, user type, and the hashed password, into our database. Similarly, if any issues arise during these steps, such as an existing username, we return a fail status along with an error message to describe the error.

### 7.2.2 Graph Generation

For our knowledge graph generation endpoints, we support the creation, retrieval, and deletion of a graph. The knowledge graph endpoint to create a graph (POST) begins by scraping and extracting the text from either a .txt file with URLs, or simply extract the text from a .zip file with HTML files or .zip file with text files. Once the text is obtained, it is posted to Stanford's Core NLP server. The Core NLP server will extract the subject-predicate-object triples from the text and send the triples as a JSON response to the Flask server. Flask then uses spaCy's natural language processor to label the subject and objects of the triples from one of four different categories: Person, Location, Organization and Misc. With the triples and the tags for each subject and object, the Flask server will build the list of nodes and query strings from the triples to be executed on the Neo4j database. Once all triples are processed, the queries to build the nodes and relationship of the graph are executed to populate the graph database. Finally, the nodes and relationships of the graph that was just created is queried on Neo4j to then return as JSON to the frontend which will render the graph with Sigma.js. This final step is what the endpoint to retrieve the graph (GET) does with the addition that a query parameter containing the graph ID is passed to specify which graph to get. The endpoint to delete a graph (DELETE) will also request the graph ID as a query parameter to delete the specified graph.

## 7.3 API

For our APIs, we have implemented RESTful APIs that facilitate communication between our React frontend and Flask backend using Axios. Axios is a JavaScript library that simplifies working with REST APIs, providing a streamlined approach to handling data exchange between the frontend and backend components of our application.

# 8 Data

## 8.1 Collections

The user will upload a collection to be processed. As mentioned the formats are a text file of URLs on each line. A .zip file containing only HTML files with content, or a .zip file of text files with content. The contents of all the collections will be processed into triples to form graph relationships.

When a collection is uploaded we store it's contents. There is a Collection directory created on the Flask level. Then a directory created of the user within Collection. Then within that directory a directory is created with the collection name and id. Then there's a collection directory where the raw collection file is stored. Then, there's a documents directory, where each file in the collection will have a separate document created and stored here as a text file regardless of the original file, such as if it's an html file it's contents will be read then stored as a text file.

With the text file of URLs, the flask server will parse the file line by line and make a request to each URL to retrieve the HTML file. The HTML file's contents will be extracted using BeautifulSoup's HTML parser.

The .zip files are unzipped by zip file library and will create a temporary directory in which it will extract the files to for processing. It will base what type of file is meant to be read by the first file that is processed. If it's an HTML file, then it will only allow for HTML files to be processed. Similarly, if it's a text file, then only text files are allowed for processing. The HTML files content will once again be read with BeautifulSoup's HTML parser and formed into triples. Likewise, the text files will be read with a file reader and processed into triples.

## 8.2 Triples

Before the triples are created, we separate the content of the documents into it's sentences and create triples one sentence at a time, as requested by the client.

The content from the collections will be processed into triples. For our triples we are using a Subject-Predicate-Object format, where the start-node is the Subject, the edge is the predicate, and the end-node is the Object. A triple is the underlying logic that we are using to form our graph. We do this by using the Stanford CoreNLP server utilizing their Open Information Extractor (OpenIE) to retrieve triples from text. For example, given the sentence:

```
John and Sally have hazel eyes.
```

This will form 2 triples as follows:

| **Subject (Start-Node)** | **Predicate (Edge)** | **Object (End-Node)** |
|---|---|---|
| John | has | hazel eyes |
| Sally | has | hazel eyes |

Table 1: Triple Extraction

Once we obtain the triples, we use Name Entity Recognition (NER) tagging. We achieve this utilizing the spaCy python library. SpaCy has their own way of tagging names, however, we are able to manipulate these tags to get the ones we want.

Using these triples, we can form the graph as follows:

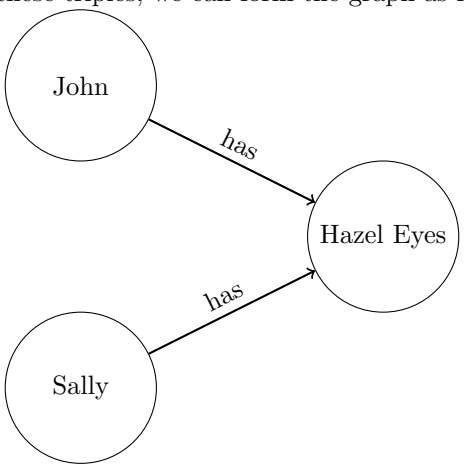


Figure 2: Graphical representation of the triples showing the relationships between subjects and objects.

| **spaCy NER Tag** | **Converted NER Tag** |
|---|---|
| PERSON | Person |
| GPE | Location |
| ORG | Organization |
| Other | Misc |

Table 2: spaCy NER tag conversion

Once we have the tags, we create a JSON representation of our triple. JSON for `John has hazel eyes.` is as follows:

```
{
    "subject" : {
        "value" : "John",
        "label" : "Person"
    },
    "predicate" : "has",
    "object" : {
        "value" : "hazel eyes",
        "label" : "Misc"
    }
}
```

There is an issue with using the Stanford CoreNLP server to form triples. It makes as many triples from the text as it can, which a lot of times can cause redundancy. This issue did not seem to diminish when removing the flag `openie.triple.all_nominals` or changing it to `openie.triple.strict`. For example the following sentence:

```
Haiti's history of national debt, prejudicial trade policies by other countries, and
 foreign intervention into national affairs contributed to the existing poverty and
     poor housing conditions that increased the death toll from the disaster.
```

Will produce the triples:

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) |
|---|---|---|
| trade policies | contributed to | poverty |
| intervention | contributed to | poverty |
| poverty | is to | existing |
| foreign intervention | contributed to | existing poverty |
| debt | is of | national |
| countries | is by | other |
| Haiti's history | contributed to | poverty |
| Prejudicial trade policies | contributed to | poverty |
| Prejudicial trade policies | contributed to | existing poverty |

Table 3: Triple Extraction

To mitigate this redundancy, we form clusters out of the triples. We form clusters using the `fuzzywuzzy` library to check the similarity of triples using the `partial_ratio` function with a threshold score of 80 where if it is equal to or above 80 it will form a cluster for that triple.

The user will have the option of selecting a simple or complex graph, based on these clusters. Due to that, we need to form a complexity score for each triple. The client requested the complexity score be based on the amount of words contained in the triple. The above triples will form two clusters of related triples, where the rest are all unique, all being scored with complexity as follows:

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) | Complexity Score |
|---|---|---|---|
| trade policies | contributed to | poverty | 5 |
| Prejudicial trade policies | contributed to | poverty | 6 |
| Prejudicial trade policies | contributed to | existing poverty | 7 |

Table 4: Cluster 1

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) | Complexity Score |
|---|---|---|---|
| intervention | contributed to | poverty | 4 |
| foreign intervention | contributed to | existing poverty | 6 |

Table 5: Cluster 2

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) | Complexity Score |
|---|---|---|---|
| poverty | is to | existing | 4 |
| debt | is of | national | 4 |
| countries | is by | other | 4 |
| Haiti's history | contributed to | poverty | 5 |

Table 6: These are unique and do not form clusters

The simple and complex graphs will be made simultaneously as the triples are produced, and then clustered. From each cluster, the simple graph will take the triples with the lowest complexity scores. Conversely, the complex graph will take the triples with the highest complexity scores. If a triple is unique and doesn't have multiple triples in the cluser, then that triple will be used for both the simple and complex. For this sentence the simple vs complex triples are as follows:

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) | Complexity Score |
|---|---|---|---|
| trade policies | contributed to | poverty | 5 |
| intervention | contributed to | poverty | 4 |
| poverty | is to | existing | 4 |
| debt | is of | national | 4 |
| countries | is by | other | 4 |
| Haiti's history | contributed to | poverty | 5 |

Table 7: Triples for Simple Graph

| Subject (Start-Node) | Predicate (Edge) | Object (End-Node) | Complexity Score |
|---|---|---|---|
| Prejudicial trade policies | contributed to | existing poverty | 7 |
| foreign intervention | contributed to | existing poverty | 6 |
| poverty | is to | existing | 4 |
| debt | is of | national | 4 |
| countries | is by | other | 4 |
| Haiti's history | contributed to | poverty | 5 |

Table 8: Triples for Complex Graph

We then form 2 lists of triples for simple triples and complex triples. Then create the nodes and edges in neo4j one triple at a time.

### 8.3 User Information

- MongoDB Atlas [6] is used to store user information like username, password, email, ID, and user type.
- A file server database will be used to store collections of zip files of text or HTML documents. Storage of user collections is yet to be implemented, so this section will be expanded upon later.

## 9 Testing

We used pytest, a Python testing framework that simplifies writing unit, functional, and integration testing. We have written user and knowledge graph tests. These consist of unit (where possible), integration and system tests. It is recommended to use VSCode with the Python Test Explorer Extension. With the extension, in VSCode, go to View->Command Palette and then type in "Python: Select Interpreter" shown in figure 3. Then click the Python Test Explorer tab on the left hand side of VSCode and configure the tests. Make sure you do this from the crisis-events-knowledge-graph-generation directory. Use the UI of Python Test Explorer to run the tests.

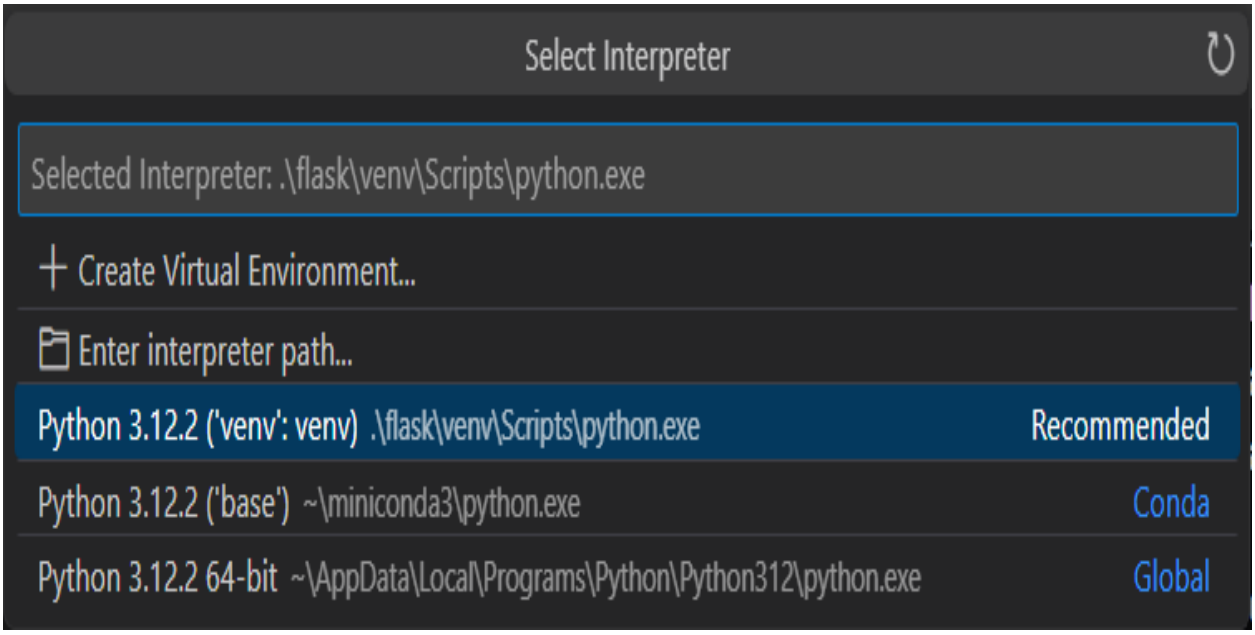


Figure 3: Selecting the ./flask/.venv/bin/python interpreter

### 9.1 Frontend

For testing our frontend, we use Playwright, a framework which allows for automated testing of web applications by performing actions such as clicking, typing, and navigation. We have chosen Playwright because it is easy to generate tests, modify those tests to make them more robust, and because we are thinking of the future teams that will continue development on our project.

We have written 3 tests for our frontend. These tests can be found in the 'react/tests/' directory. Our tests for the frontend cover:

- Logging In
- Signing Out
- Creating a New Account
- Creating a New Collection
- Deleting a Collection
- Toggling On/Off the Simple/Complex Graph Toggle

These tests cover the most important features of our UI. There are other aspects of our UI we cannot test through an automated test, such as actually looking at a graph. These are things which are better assessed through visual inspection to ensure the graphical elements render correctly and are visually appealing.

To run our all of our frontend tests: follow the instructions below. Note we are assuming you are able to run the project normally as described in this report before you attempt to run our tests.

1. Start the Flask Backend:
    - Navigate to the project's 'flask' directory
    - In one terminal, run the command:

```
flask --app ./src/app.py run -p 5000
```

NOTE: This is assuming you have properly set your environment up for running the backend! Check the Developer Manual.

2. Start the Stanford CoreNLP:

   - In another terminal, navigate to the directory where you have the Stanford CoreNLP installed and run the standard command needed to execute it:

     ```
     java -Xmx4g -cp "*"
     edu.stanford.nlp.pipeline.StanfordCoreNLPServer -
     serverProperties StanfordCoreNLP.properties -preload
     tokenize,ssplit,pos,lemma,ner,parse,depparse,natlog,openie -
     status_port 9000 -port 9000 -timeout 15000 -
     openie.triple.all_nominals true
     ```

3. Run Playwright in the Frontend:

   - In a third terminal, navigate to the ‘react‘ directory in the project.
   - To run the ‘create account‘ test, run the command:

     ```
     $ npx playwright test tests/user/create_account.spec.js --
     headed --project=create_account
     ```

   - To run the ‘create collection‘ test, run the command:

     ```
     $ npx playwright test tests/user/create_collection.spec.js --
     headed --project=create_collection
     ```

   - To run the ‘user login‘ test, run the command:

     ```
     $ npx playwright test tests/user/user_login.spec.js --headed
     --project=user_login
     ```

# 10 User Manual

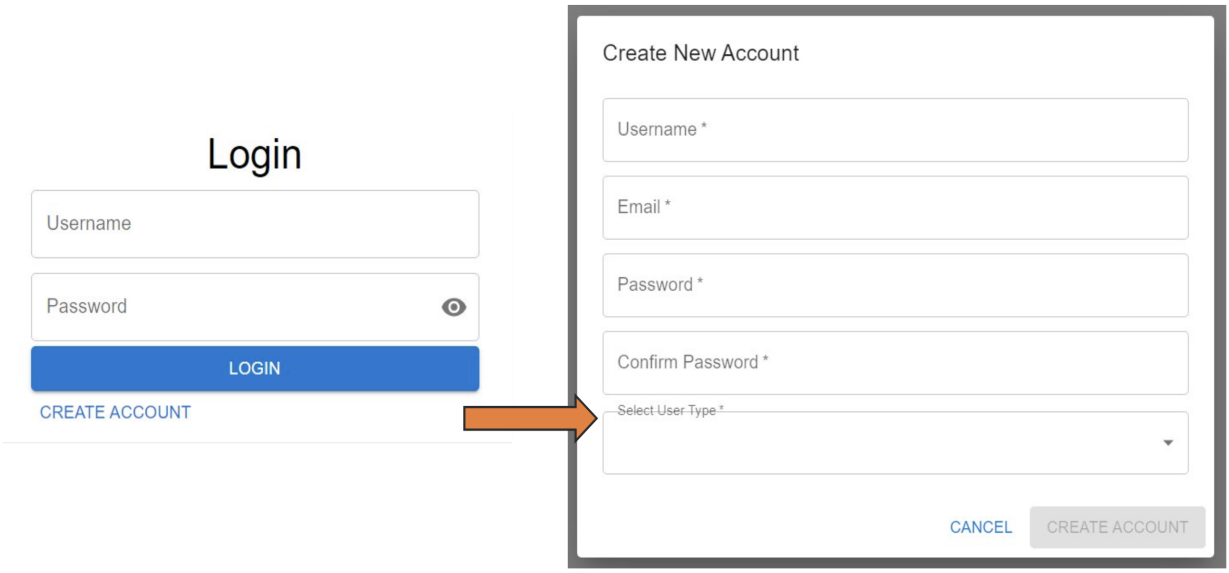


Figure 4: Login Page

Users are greeted with a login screen where they can create an account or login using their username and password.

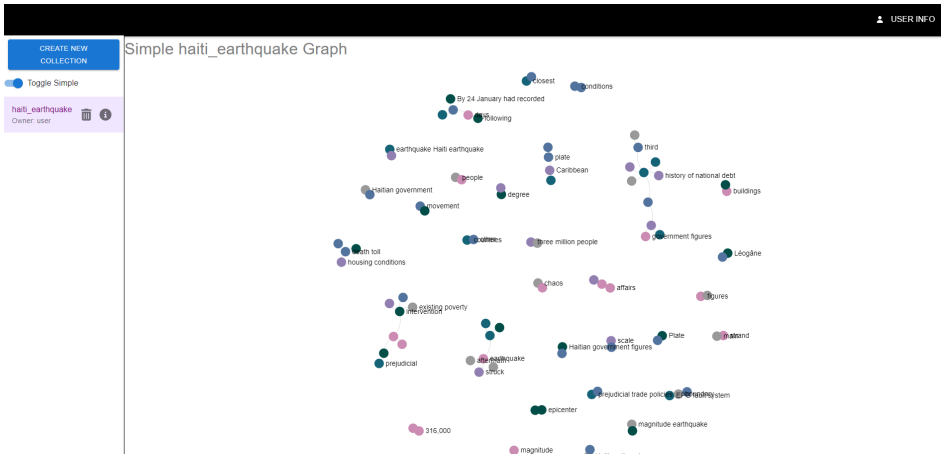


Figure 5: Main Page

Once logged in, current collections are displayed in a side menu on the left. Selecting a collection from the side menu displays the knowledge graph corresponding to that collection. Clicking the trash icon on a collection brings up a modal that asks if the associated collection should be deleted or not. Confirming to delete the collection from this modal will remove the collection from the side menu and its associated data from the backend. Clicking the information button next to the trash button causes a modal to pop up with collection information such as the titles of documents supplied. Clicking the user info button in the top right corner will cause a modal with user information to pop up. This modal has the option for the user to logout. At the top of the side-menu is a "Create New Collection" button. If clicked, this opens a modal prompting

the user to insert the name of the new collection they are creating and a file selector asking them to input a file to be sent to the backend to be used to generate a knowledge graph.

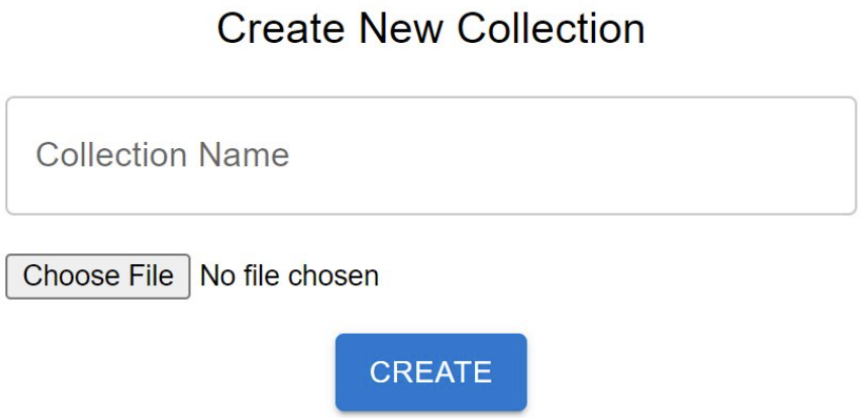


Figure 6: Create Collection

A name for the collection can be entered, and a file can be chosen. The file can be a text file of URLs with one URL per line, a zip file of HTML documents, or a zip file of text documents. Click create to store the documents and begin generating a graph.

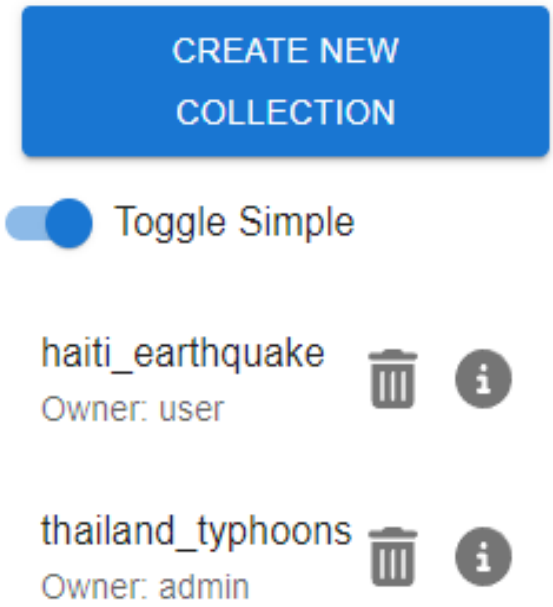


Figure 7: Sidebar

Once the graph is generated, the user will have the option to select the graph from the menu in the sidebar, information about the graph, and the complexity of the graph. If a graph is selected selected, the graph will display in the center of the user's screen and the user will be able to see information relevant to their input documents. They can interact with the graph by dragging it around and zooming in and out. The user will also be able to see who created the collection under the collection item's name.

If the simple graph option is chosen, the user will be presented with a simplified version of the knowledge graph. This graph has less nodes, and subject names are less complicated.

Simple haiti_earthquake Graph

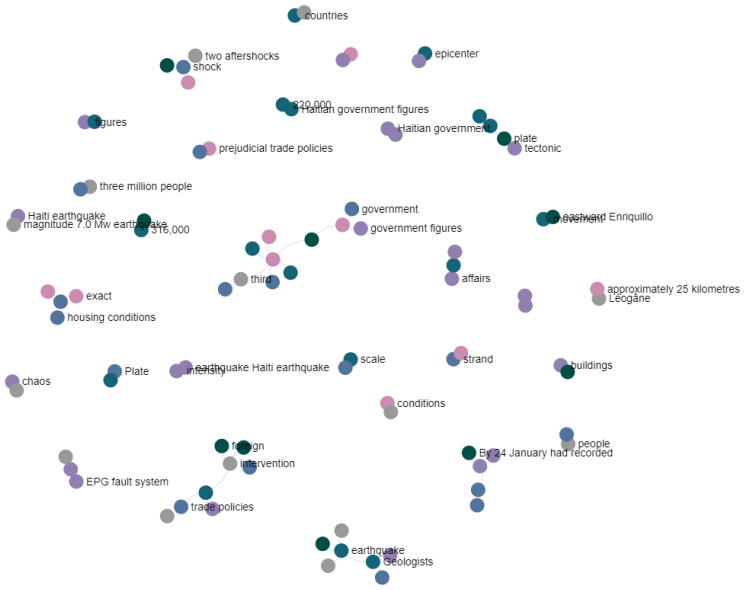


Figure 8: Simple Graph

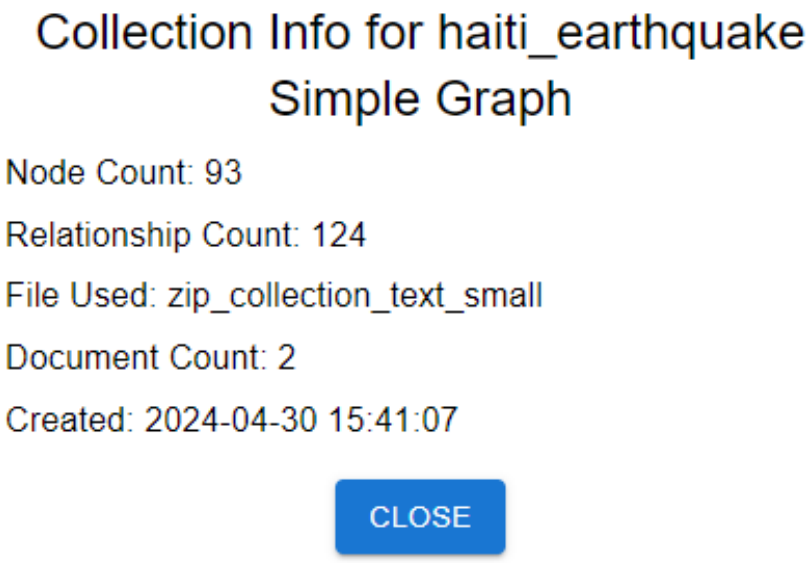


Figure 9: Simple Graph Information

If the complex graph option is chosen, the user will be presented with a complex version of the knowledge graph. This graph has more nodes, and subject names will be more detailed and complex.


Figure 10: Complex Graph

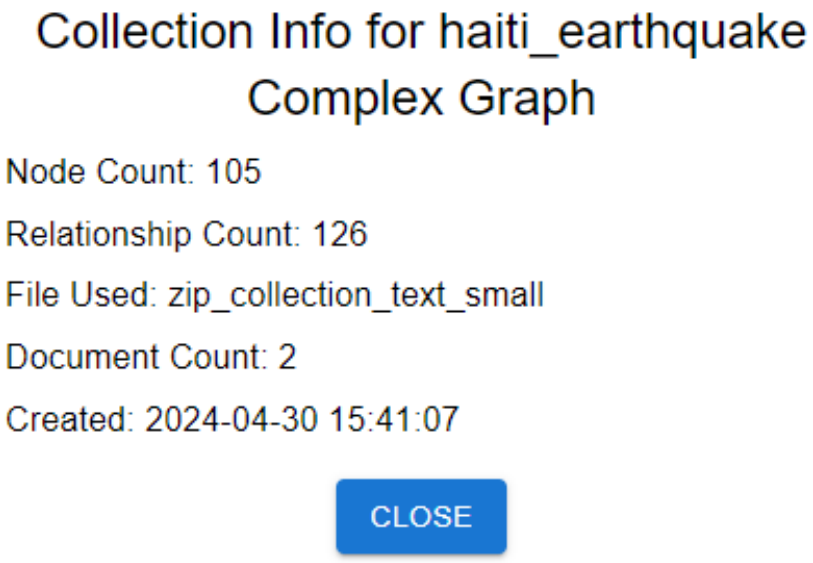


Figure 11: Complex Graph Information

# 11 Developer Manual

The recommended operating system for this application are Linux and Windows with a preference towards **Windows**. Due to permissions issues, it is not recommended to setup the application with Mac.

## 11.1 Cloning the Repository

Install git for your operating system. You can find the repository for the project at: https://git.cs.vt.edu/mattwalters/crisis-events-knowledge-graph-generation.

clone the project locally using **git clone** followed by the the ssh or https link
**SSH**: git@git.cs.vt.edu:mattwalters/crisis-events-knowledge-graph-generation.git
**HTTPS**: https://git.cs.vt.edu/mattwalters/crisis-events-knowledge-graph-generation.git

cd into the crisis-events-knowledge-graph-generation directory

## 11.2 Prerequisites

For both local dev and production deployments:

Install VSCode
Install Docker

It is recommend to install a Docker GUI such as Docker Desktop:

- Windows: https://docs.docker.com/desktop/install/windows-install/
- Linux: https://docs.docker.com/desktop/install/linux-install/
- Mac: https://docs.docker.com/desktop/install/mac-install/

Or the Docker VSCode extension.

### 11.2.1 Running in a local deployment (recommended)

Install at least Python 3.12.
In a terminal, check your python version with the following command (Note: if your Python CLI is set up to the alias "python3" then replace "python" in the following command with "python3"):

```
python --version
```

If you have multiple python versions installed, you will have to modify your PATH variable to point to the python executable for python 3.12. Alternatively, with Windows you can use the python launcher to run python 3.12 by replacing "python" in the commands displayed in the following sections with "py -3.12". Check that python 3.12 is installed for the python launcher by running "py --list" in a powershell terminal.

Install at least Node.js v18.17.1.
Prebuilt Node.js installers for Windows and Mac are located here.
Prebuilt Node.js binaries for Windows, Mac and Linux are located here.
The redistributables for Node.js v18.17.1 are available here. In a terminal, check your node version with the following command:

```
node -v
```

Install at least npm version 9.6.7 (npm comes with Node):

```
npm install -g npm@9.6.7
```

In a terminal, check your node version with the following command:

```
npm -v
```

Follow directions in the sections Local Dev Development - Backend, Local Dev Development - Frontend, Local Dev Deployment - Overview and Local Dev Deployment - Tearing Down the Application.

### 11.2.2 Running with a fully containerized deployment

Note that the docker deployment is **not integrated to run with Mac** at this time. (mac requires Docker Desktop which uses a docker compose version that is too low)

Install Docker Compose (Docker compose comes with Docker Desktop, but you may have to install it as a plugin on Linux). Docker compose should have a version of at least v2.26.1.

Check this by running the following command:

```
docker compose version
```

Follow the directions of the section Running the Application with Docker

## 11.3 Local Dev Deployment - Backend

### 11.3.1 Note for Windows

Use a PowerShell terminal. In your current PowerShell terminal or process you may have to run the following command to allow execution of a PowerShell script:

```
Set-ExecutionPolicy Bypass -Scope Process
```

This command allows PowerShell scripts to run without being blocked by the execution policy set on the Windows System. Whenever you create a new powershell terminal you will need to run the command above.

### 11.3.2 Environment Variables

To set up the .env file in the /flask/src directory, there is a script supported for Windows, Linux and Mac. In a new terminal, cd to /flask.

**Windows**
In the flask directory run the following command to set up the environment variables (Note: if your Python CLI is set up to the alias “python3” then replace ”python” in the following command with “python3”):

```
python scripts/setup_env.py
```

**Linux and Mac**
In the flask directory run the following command to set up the environment variables (Note: if your Python CLI is set up to the alias “python3” then replace ”python” in the following command with “python3”):

```
python scripts/setup_env.py
```

### 11.3.3 Check that the Docker Daemon is Running

To check if the docker daemon is running, in a terminal run the following command:

```
docker ps
```

If you encounter an error message similar to this: "error during connect: this error may indicate that the docker daemon is not running:" then you must start the docker daemon. A simple solution is to start docker desktop. Reference the docker docs for other methods of starting the docker daemon.

### 11.3.4 Flask

To start up the flask server, **it is recommended to do the manual setup**.There are scripts supported for Windows and Linux for future developers, but some tweaking may be needed.

#### Manual Setup

##### Windows

1. From the top-level directory of the repository (crisis-events-knowledge-graph-generation/), change to the flask directory in a new terminal:

   ```
   Set-Location .\flask\
   ```

   Ensure your terminal is in the flask directory for the following commands.

2. Ensure pip is installed (Note: if your Python CLI is set up to the alias "python3" then replace where it is says "python" in the following commands with "python3"):

   ```
   python -m ensurepip
   ```

3. Install the virtualenv module:

   ```
   python -m pip install virtualenv
   ```

4. Create the virtual environment directory in the .venv folder:

   ```
   python -m virtualenv .venv
   ```

5. Activate the virtual environment for your terminal:

   ```
   .venv\Scripts\activate.ps1
   ```

6. Install all the modules in your virtual environment from requirements.txt:

   ```
   pip install -r requirements.txt
   ```

7. Run these commands in your virtual environment:

   ```
   pip uninstall pymongo -y; pip install pymongo
   python -m spacy download en_core_web_sm
   python -m spacy download en_core_web_lg
   ```

8. Once the Neo4j (section 11.3.5), Mongo (section 11.3.6) and Stanford CoreNLP (section 11.3.7) servers are running, start the flask server:

```
flask --app ./src/app.py run -p 5000
```

**Linux and macOS**

1. From the top-level directory of the repository, change to the flask directory:

   ```
   cd ./flask/
   ```

   Ensure your terminal is in the flask directory for the following commands.

2. Ensure pip is installed (Note: if your Python CLI is set up to the alias "python3" then replace where it is says "python" in the following commands with "python3"):

   ```
   python -m ensurepip
   ```

3. Install the virtualenv module:

   ```
   python -m pip install virtualenv
   ```

4. Create the virtual environment directory in the .venv folder:

   ```
   python -m virtualenv .venv
   ```

5. Activate the virtual environment for your terminal:

   ```
   . .venv/bin/activate
   ```

6. Install all the modules in your virtual environment from requirements.txt:

   ```
   pip install -r requirements.txt
   ```

7. Run these commands one by one:

   ```
   python -m pip uninstall pymongo -y; python -m pip install pymongo
   python -m spacy download en_core_web_sm
   python -m spacy download en_core_web_lg
   ```

8. Once the Neo4j (section 11.3.5), Mongo (section 11.3.6) and Stanford CoreNLP (section 11.3.7) servers are running, start the flask server:

   ```
   flask --app ./src/app.py run -p 5000
   ```

**Setup Scripts**

These scripts are more likely to produce errors than manual setup.

- **Windows**
  In the flask directory run the following command to set up the python virtual environment and start the flask server. Make sure you are using a new terminal:

  ```
  scripts\start_flask.ps1 -s
  ```

- **Linux**
  In the flask directory run the following command to set up the python virtual environment and start the flask server. Make sure you are using a new terminal:

  ```
  . scripts/start_flask.sh -s
  ```

#### 11.3.5 Neo4j

To install Neo4j on your operating system instead of through docker follow the directions listed here. Scroll down to the "Graph Database Self-Managed" section, select "Community" and then download the package for your operating system.

To start the Neo4j container:

1. Ensure that docker is installed and running in your environment. Run the commands below on a new terminal.
2. If you are on Windows run the start_neo4j.ps1 script located in the flask directory with the following command:

   ```
   .\scripts\start_neo4j.ps1
   ```

   If you are on Linux or macOS run the start_neo4j.sh script located in the flask directory with the following command:

   ```
   ./scripts/start_neo4j.sh
   ```

   Both scripts create a neo4j container that runs a local neo4j database server. The data for this server is stored in a docker volume named "neo4jdbdata". The output of these scripts will be an active log of the neo4j database server.
3. Check that a container instance of the neo4j image is running by running the following command:

   ```
   docker container ps
   ```
4. On your computer in a browser, navigate to http://localhost:7474/. You should be automatically logged in. Within the .\start_neo4j.ps1 and .\start_neo4j.sh there is a comment to explain how to enable authentication. You have to remove the "–env=NEO4J_AUTH=none" from the docker run command within the scripts, navigate to http://localhost:7474/, sign in with the default username and password (default username: neo4j, default password: neo4j) and then set your password.

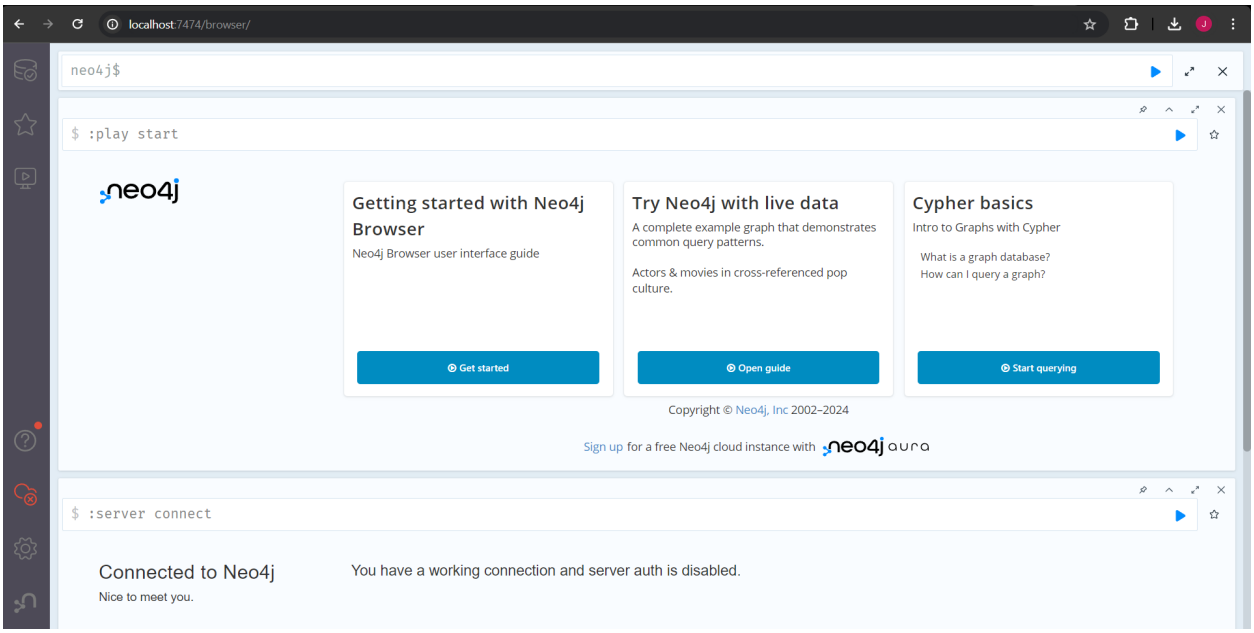


Figure 12: Database Administrator View of Local Neo4j database

### 11.3.6 Mongo

If you want to run MongoDB as a server on your operating system instead of through docker, follow the directions listed here.

To start the Mongo container, cd into the /flask directory:

1. Ensure that docker is installed and running in your environment. Run the commands below on a new terminal.

2. If you are on Windows run the start_mongo.ps1 script located in the flask directory with the following command:

   ```
   .\scripts\start_mongo.ps1
   ```

   If you are on Linux or macOS run the start_mongo.sh script located in the flask directory with the following command:

   ```
   ./scripts/start_mongo.sh
   ```

   Both scripts create a mongo container that runs a local mongo database server. The data for this server is stored in a docker volume named "mongodbdata".

3. Check that a container instance of the mongo image is running by running the following command:

   ```
   docker container ps
   ```

### 11.3.7 Stanford CoreNLP

The Stanford CoreNLP Server is a crucial part of the natural language processing and knowledge graph creation. We have provided options for running the server as an image locally or as a docker container. **Running the image locally is preferred** over the container option as the local server was noticeably quicker over the container when processing triples.

**Running Stanford CoreNLP as a Local Server (Recommended)**

1. Make sure you have Java installed. Open a terminal and type

   ```
   java -version
   ```

2. If java is installed, you'll see something like

   ```
   java version "17.0.9" 2023-10-17 LTS
   ```

3. If java is not installed, you'll see something like

   ```
   java: command not found
   ```

4. If you get this then install Java keep in mind this has been tested with Java version 17.0.9 so it's not guaranteed to work with other versions.
5. Download the Stanford CoreNLP
6. Unzip the .zip file
7. Using a new terminal, change directory to: cd ./stanford-corenlp-4.5.6/stanford-corenlp-4.5.6
8. Start the server with the command:
   (a) macOS, Git Bash, or Unix like Systems

   ```
   java -Xmx4g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer \
   -serverProperties StanfordCoreNLP.properties \
   -preload tokenize,ssplit,pos,lemma,ner,parse,depparse,natlog,openie \
   -status_port 9000 -port 9000 -timeout 15000 -openie.triple.all_nominals true
   ```

   (b) Windows

   ```
   java -Xmx4g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer ^
   -serverProperties StanfordCoreNLP.properties ^
   -preload tokenize,ssplit,pos,lemma,ner,parse,depparse,natlog,openie ^
   -status_port 9000 -port 9000 -timeout 15000 -openie.triple.all_nominals true
   ```

**Running Stanford CoreNLP as a Docker Container**

**Windows**
Cd into /flask and run the following script to start the stanford corenlp docker container:

```
.\scripts\start_stanford_corenlp.ps1
```

**Linux or Mac**
Cd into /flask and run the following script to start the stanford corenlp docker container:

```
./scripts/start_stanford_corenlp.sh
```

Check that a container instance of the stanford corenlp image is running by running the following command:

```
docker container ps
```

## 11.4 Local Dev Deployment - Frontend

### 11.4.1 React

1. Make sure you have at least NPM version 9.6.7 and Node version v18.17.1 installed using **npm –version** and **node –version**.
2. In the codebase and in a new terminal, use cd to navigate to /react
3. In /react, run `npm install --force` to install dependencies
4. DO NOT WORRY ABOUT VULNERABILITIES FOUND! npm audit is super broken, and running the recommended "npm audit fix –force" command seen in the console is very likely to break things. This is a well-known issue in the React community. For further reading, check `https://overreacted.io/npm-audit-broken-by-design/`
5. Also in /react, run "npm start" to start the React app, opening the UI in your web browser
6. All dependencies *should* be installed, however, if some that are not installed, this will result in React errors when viewing the page. You will know if this happens because the webpage will be red and have errors appearing. In this case, simply install whichever module is not found or installed by running "npm install [module name] –force", where "[module name]" would be something like "graphology". Ex: "npm install graphology –force".

## 11.5 Local Dev Deployment - Overview

When all portions of the Local Dev Deployment - Backend (section 11.3) and Local Dev Deployment - Frontend (section 11.4) sections are completed you should have the following servers running on your machine:

1. Flask server running in a terminal on port 3000
2. Mongo server running in a docker container or as a daemon process on your machine on port 27017
3. Neo4j server running in a docker container or as a demon process on your machine on ports 7474 and 7687
4. Stanford CoreNLP server running in a terminal or in a container on port 9000
5. React server running in a terminal on port 5000

Navigate to http://localhost:3000 in your web browser (such as Google Chrome) to use the application!

## 11.6 Local Dev Deployment - Tearing Down the Application

1. Stop the react server from the terminal it is running in.
2. Stop the flask server from the terminal it is running in.
3. Stop the stanford corenlp server. If the server is running in a terminal, then stop the server from the terminal. If it is running in a container run the following commands (Assuming you started the container from the start_stanford_corenlp .sh or .ps1 scripts) :

```
docker container stop corenlp
docker container rm corenlp
```

4. Run the following commands (Assuming you started the mongo and neo4j containers from the start_mongo and start_neo4j .sh or .ps1 scripts). The data of the mongo and neo4j database servers are stored on the docker volumes named "mongodbdata" and "ne4jdbdata" respectively:

```
docker container stop mongo
docker container rm mongo
docker container stop neo4j
docker container rm neo4j
docker volume rm mongodbdata
docker volume rm neo4jdbdata
```

5. In the flask directory, delete the "Collections" folder

## 11.7 Local Production Deployment - Running the Application with Docker

### 11.7.1 Docker Stack overview

You can build and deploy the application via docker. This application has 5 containers that are deployed via docker compose. The containers communicate across a docker network named: "knowlegde_graph_network". The 5 containers are:

1. **Neo4J Database Server**. This contains nodes and relationships for each graph. You can view this container in the browser at `http://localhost:7474/`
2. **MongoDB Database Server**. This contains the user and collection information. Exposed within the knowlegde_graph_network on port 27017.
3. **Stanford CoreNLP Server**. Exposed within the knowlegde_graph_network on port 9000.
4. **Flask Backend Server**. Runs the flask application. Exposed on port 5000
5. **React Frontend Server**. Runs the react application. Exposed on port 3000.

### 11.7.2 Deploying the Docker Stack

Ensure you have docker compose installed. docker compose comes with Docker desktop though you may have to install the plugin with Linux (Link below). Run the following command to check if docker is installed:

```
docker compose --help
```

If it lists the help page for docker compose then docker compose is installed. If it says that the command is not found then docker compose is not installed. Refer to the official docker instructions to install docker compose for your docker service instance. To deploy the application navigate to the top level directory of the cloned repository: "/path/to/crisis-events-knowledge-graph-generation". Run the following command:

```
docker compose up -d --build
```

If you are running this for the first time, it will take a while as it must pull the neo4j, mongo and stanford core nlp images from docker hub and then build the backend and frontend containers. Once all containers are running (Shown in figures 13 and 14), you must wait until the backend server is ready to accept requests. This is because backend server must wait for neo4j, mongo and stanford core nlp servers to be

started and ready to accept requests as well. Almost always, the neo4j and stanford core nlp servers take longer to start listening on their servers than the backend. Thus, the backend has been configured to wait for these servers (including mongo) to be ready before starting its own server. The backend waits up to 5 minutes for each of the servers to be up in the following order: mongo, neo4j, stanford corenlp. Check the logs for the frontend container. Once it says "INFO Accepting connections at http://localhost:3000" then the frontend is ready. Check the logs for the backend container. Once it says "INFO:waitress:Serving on http://0.0.0.0:5000" (Shown in figures 15 and 16) then the application is ready to use!

If you want to remove the docker volumes that store the database data (neo4j, mongo and file server data) of the application (Fully resetting the application) then run

```
docker compose down -v
```

To stop the application and keep the database data of the run the following docker command:

```
docker compose down
```

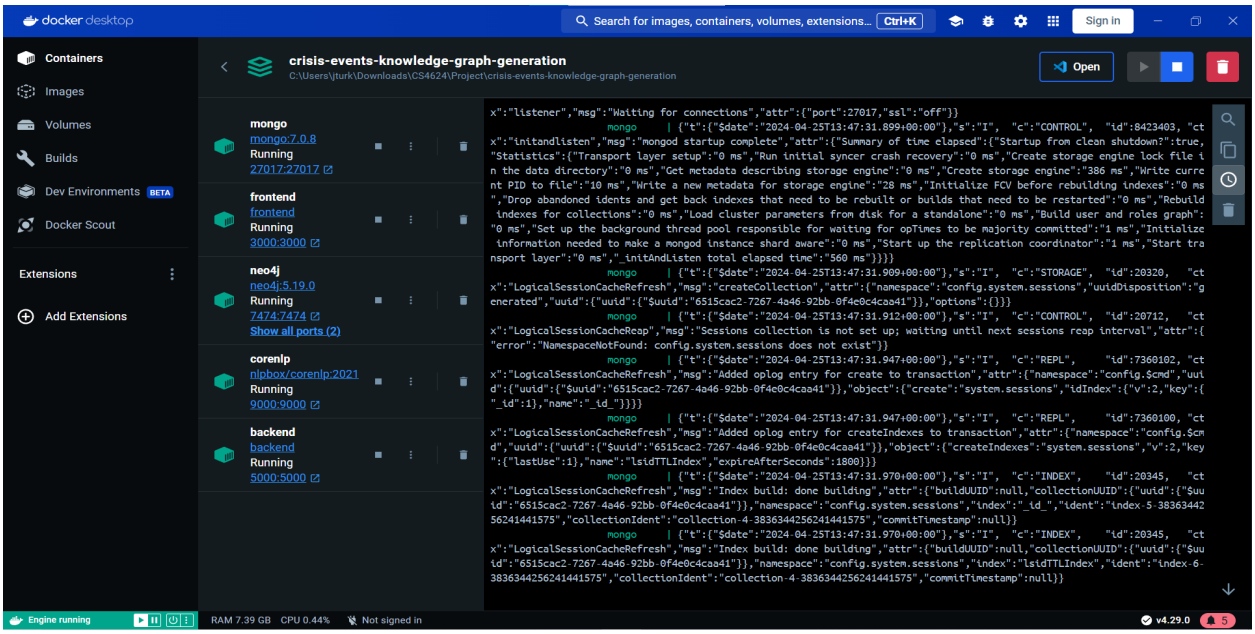


Figure 13: View of the application deployed in a docker stack on Docker Desktop

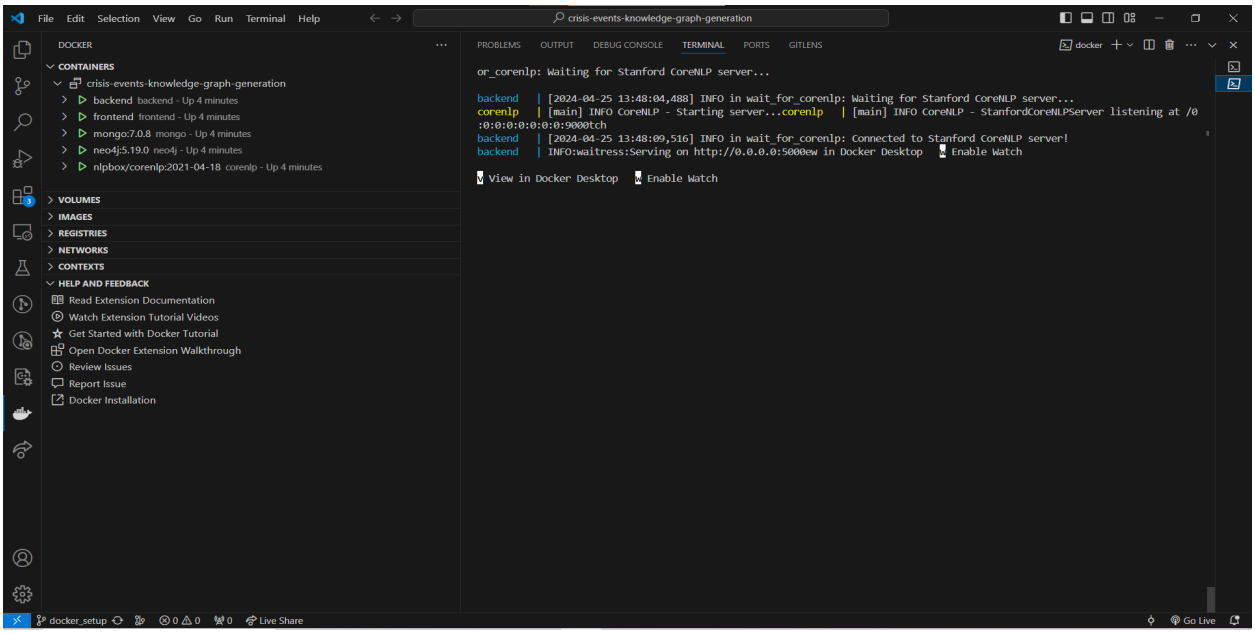


Figure 14: View of the application deployed in a docker stack on VSCode using the VSCode Docker Extension

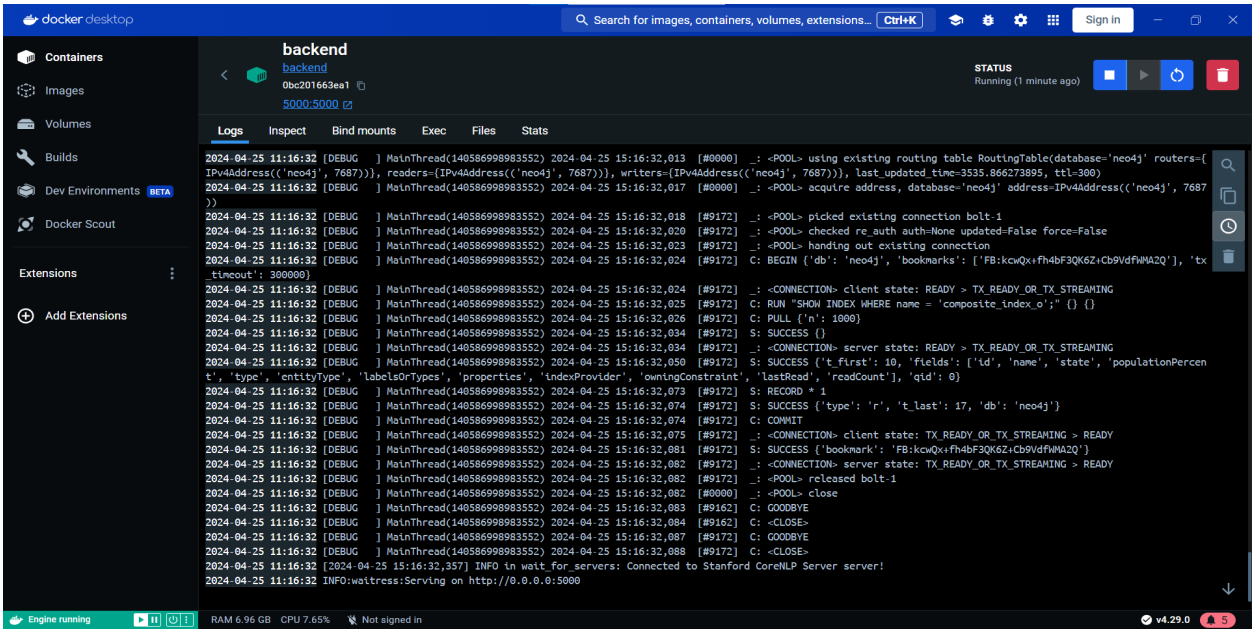


Figure 15: Logs of backend container showing that the flask server is ready in Docker Desktop

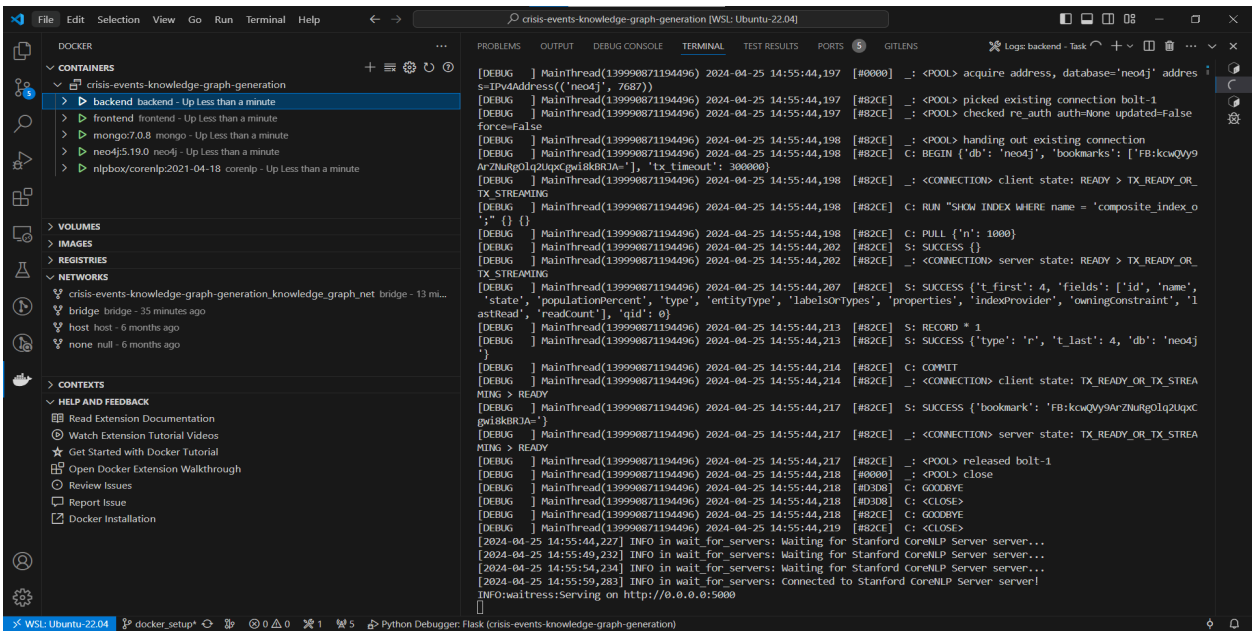


Figure 16: Logs of backend container showing that the flask server is ready in VSCode

## 11.8 Project File Structure

The project is split into 2 separate folders: "flask" and "react" representing the backend and frontend respectively.

### 11.8.1 Top Level File Structure

/crisis-events-knowledge-graph-generation/ is the top level directory of the cloned git repository.

```
/crisis-events-knowledge-graph-generation/
├── flask/ - Flask backend directory
├── react/ - React frontend directory
├── .gitignore - Ignores folders and files so git does not commit them
├── docker-compose.yml - Creates the docker stack to deploy the application.  Run docker compose
│   up -d --build in the /crisis-events-knowledge-graph-generation to build and deploy the
│   application
└── README.md - Describes the /crisis-events-knowledge-graph-generation code structure
```

### 11.8.2 Flask File Structure

ets user's collections from file storage. .3 tests/ - Test files . .4 __pycache__/ - Stores bytecode of python modules to speed up deployment of the flask server . .4 .pytest_cache/ - Used to analyze failures from pytest runs. Caches pytest results . .4 test_data/ - Sample data used in knowledge_graph_test.py as files uploaded to tests that run the /knowledge_graph endpoint for . .4 __init__.py - For importing and setting up the tests directory as a module. . .4 knowledge_graph_test.py - Contains tests for the /knowledge_graph endpoint and tests the /users_/collections endpoint . .4 root_test.py - Tests the / url endpoint. .4 users_test.py - Tests the endpoints for /users such as /users/login and /users/create. .3 .dockerignore - Ignores certain files and folders from the current /flask directory so docker does not have to remember them in the build process. .3 .gitignore - Ignores folders and files so git does not commit them. .3 backend_dockerfile - Creates the docker image for the backend. .3 README.MD - Describes the flask backend code structure. .3 requirements.txt - Lists the python packages to install. Is used for installing these python packages when creating a python virtual environment .

```
flask/ - Flask backend directory
    ├── .pytest_cache/ -
    ├── .venv/ - Contains the python virtual environment that installs the required python
    │   packages and contains the python executable to run the flask application
    ├── Collections/ - File Server Database folder(Created when first collection is created)
    │
    ├── sample_data/ - Sample data used for testing
    ├── scripts/ - Various scripts to run and support the backend application
    │   ├── __pycache__/ - Stores bytecode of python modules to speed up deployment of the flask
    │   │   server
    │   ├── index_creation.py - Creates the composite index stored in the Neo4j database to improve
    │   │   the speed of matching nodes during relationship creation
    │   ├── setup_env.py, .sh & .ps1 - Creates .env file in src for the flask application to
    │   │   use .4 start_flask.sh & .ps1 - Sets up the flask developer server environment
    │   ├── start_mongo.sh & .ps1 - Starts the mongo database container
    │   ├── start_neo4j.sh & .ps1 - Starts the neo4j database container
    │   ├── wait_for_servers.py - Contains function to wait for the mongo and stanford corenlp
    │   │   servers to be running (index_creation.py waits for neo4j server)
    ├── src/ - Source code
    │   ├── __pycache__/ - Stores bytecode of python modules to speed up deployment of the flask
    │   │   server
    │   ├── __init__.py - For importing and setting up the src directory as a module
    │   ├── .env - File is created from scripts/setup_env.py.  Stores environmental variables
    │   │   that the flask server uses
    │   ├── app.py - Creates the Flask app and registers the urls for the app
    │   ├── collection_processor.py - Scrapes urls and extracts text from files for the uploaded
    │   │   collection
    │   ├── knowledge_graph_utils.py - Contains the CQL (Cypher Query Language) query code to
    │   │   process, create, get and delete the nodes and relationships from a graph in the
    │   │   Neo4j database
    │   ├── root.py - Code for the root / url endpoint
    │   ├── triple_extraction.py - Gets the triples from the stanford corenlp server and extracts
    │   │   the data
    │   ├── triple_testing.py - Tests triple extraction
    │   ├── users.py - Login and create users
```

### 11.8.3 React File Structure

```
react/ - React frontend directory
    ├── node_modules/ - Modules installed to run the react server
    ├── playwright-report - Contains html reports of the playwright test runs
    ├── public/ - Static files that are not processed by webpack
    │   ├── favicon.ico - React logo
    │   ├── index.html - Base html page
    │   ├── logo192.png - React image
    │   ├── logo512.png - React image
    │   ├── manifest.json - Metadata about the web application
    │   └── robots.txt - Specifies whether a web crawler can access the website
    ├── src/ -
    │   ├── components/ - React components rendered on the website
    │   │   ├── graph/ - Graph components
    │   │   │   ├── KnowledgeGraph.js - Contains the knowledge graph display box whee SigmContainer
    │   │   │   │   is placed
    │   │   │   ├── NavBar.js - Displays the navigation bar in the home page
    │   │   │   └── SigmaContainer.js - Component where the knowledge graph is displayed
    │   │   ├── modals/ - Modal components
    │   │   │   ├── CollectionInfo.js - Component that displays the information of each collection
    │   │   │   │
    │   │   │   ├── CreateModal.js - Component that displays the information to create a collection
    │   │   │   │
    │   │   │   ├── DeleteModal.js - Component to delete a collection
    │   │   │   ├── LoginModal.js - Component to login as a user
    │   │   │   ├── SignUpModal.js - Component to sign up as a user
    │   │   │   └── UserInfoModal.js - Component that displays the logged in user's information
    │   │   │
    │   │   └── sidemenu/ - SideMenu components
    │   │       ├── Collection.js -
    │   │       └── SideMenu.js -
    │   ├── pages/ - Pages that can be accessed and displayed on the website
    │   │   ├── Login.js - User login and create account page
    │   │   └── MainPage.js - Main/home page of the website.  Where knowledge graph, collection
    │   │       and user info are displayed
    │   ├── styles/ - Custom styles used on the website
    │   │   └── login.css - Styles for the login page
    │   ├── App.css - Styling for App.js
    │   ├── App.js - Root of the react application
    │   ├── constants.js - Constants used throughout the react code
    │   ├── index.css - Styling for the index.html page
    │   ├── logo.svg - Svg logo
    │   └── reportWebVitals.js - Measures performance of application
    ├── tests/ -
    ├── .dockerignore - Ignores certain files and folders from the current /react directory
    │   so docker does not have to remember them in the build process
    ├── .gitignore - Ignores folders and files so git does not commit them
    ├── package-lock.json - Locks the node dependencies versions
    ├── package.json - Lists the node dependencies that need to be downloaded and installed
    │   to run the react application
    ├── playwright.config.js - Configuration options for the tests/ folder that uses playwright
    │
    └── README.md - Describes the /react code structure
```

# 12 Deliverables

Our deliverables consist of the base code for our application which includes frontend, backend, scripts, and any files needed for it to run.

## 12.1 Frontend

This consists of React.js and JavaScript code required to run the frontend. Steps to start the frontend can be found in the developer's manual. Frontend tests are found here too, and steps can be found in the testing section.

## 12.2 Backend

This consists of primarily Python code for our Flask backend required to run the backend. Steps to start the backend can be found in the developer's manual.

## 12.3 Scripts

The majority of scripts that set up necessary variables and deploy containers can be found in flask/scripts. Our deployment scripts help to simplify the process of deploying our MongoDB and Neo4j databases that our application uses for storage. Our setup scripts for flask and environment variables help to set up any necessary variables used in our code and to install any dependencies that our backend requires.

# 13 Lessons Learned

A big challenge we had was getting everything to work together. We each assigned ourselves to specific domains of the project and gave each other updates as to our progress. Once we were able to get our different components up and running, the real challenge appeared which was integrating all of these individual parts to form our application. We found it difficult to coordinate what each component needed from other components and how they interacted with each other. This led to more time spent researching how to effectively integrate these different components into our final application. The lesson we learned from this experience is that in a project like this, the team should work more closely together with each other and integrate piece by piece iteratively to minimize the number of integration issues faced. Below we will get into specifics in challenges we faced.

## 13.1 Challenges & How We Tackled Them

### 13.1.1 Natural Language Processing (NLP)

One challenge we faced was that our project must be able to perform some tasks with Natural Language Processing (NLP). In particular, the NLP we dealt with was triple extraction and NER tagging. There are many resources and libraries to help with NLP. Nonetheless, NLP is a complex subdivision of computer science. It deals with complex machine learning models based on human languages for the computer to understand written text. Due to the complexity, it took a lot of trial and error to find the right resources to work for our project. We tried machine learning models from hugging face that can perform NLP. The problem with these, were they were quite large and took up a lot of computer resources as well as inconsistent in their results. In order to fine tune the models to work properly for our task, we would've had to train the models ourselves to get it to perform exactly what we want. However, to train a model it would be even more resource intensive. Basically, the machine learning models were far too complex for what we were trying to do, and couldn't be narrowed down to work properly for our goal. Conversely, we tried using NLP python libraries such as nltk and cltk.triple_extraction. Unfortunately, these were too basic, and couldn't come up with proper triple extraction. Luckily, we eventually found the Stanford CoreNLP server which has the Open Information Extraction or OpenIE that is specialized specifically for triple extraction.

### 13.1.2 General Database Setup

One of the challenges we encountered during our project was determining the most suitable MongoDB database deployment method for storing user information. We needed to balance considerations such as storage limitations and deployment simplicity. Initially, we considered deploying MongoDB locally and utilizing a Docker container. However, this approach would have required extensive setup on the user's end and familiarity with integrating databases into containers. We first opted to use MongoDB Atlas [6], a cloud-hosted version of MongoDB. This decision eliminated the need for users to set up and manage local database environments. However, we then transitioned back to a local MongoDB setup using containers once we finalized the code. This allowed our code to be deployable in any environment setup along with no space restrictions.

### 13.1.3 Knowledge Graph Generation

There were initial challenges with choosing the Graph DBMS for the project. We decided on Neo4j due to its development environment and setup, integration with Python, and rich feature support [1]. Since no one on the team had any experience with graph databases, we were not sure which graph database to choose based on what we needed. Extensive research was done to decide on using Neo4j for our project.

Another challenge was learning the query language that interacts with Neo4j. Neo4j has its own custom-built query language called Cypher Query Language (CQL) [3]. This query language is optimized to support queries for performing CRUD operations on nodes and graphs within the graph database. We had to learn the syntax and semantics of the language in order to perform the advanced queries that we needed to generate, get, and delete the nodes and relationships from the graph. This learning curve increased the time it took to develop the code that stores and queries the data from the Neo4j graph database.

Because of the large number of nodes and relationships we will produce for each graph, efficient CQL queries need to be used to hasten the speed of posting the data to the graph database. This required additional research to find queries that support batch creation of nodes and optimization steps to the code itself to reduce the runtime of the graph generation algorithm [2].

### 13.1.4 Triplestore JSON Format

Initially, we had difficulty in deciding on the JSON format that the triples will be outputted as from the CoreNLP model. We wanted to have separate arrays for the nodes and relationships with a third triplestores array that references the index of the nodes and relationships from the other two arrays. However, doing this would make it hard to read the JSON file. We then went with a JSON that has an array of the subject-predicate-object triplets. This format worked, but it was inefficient as the triplets have to be continuously appended to the array and then the knowledge graph iterates through the triplet array again to build the relationship query string to execute on the Neo4j database. We decided we would build the relationship query string and the list of nodes as the triplets are being traversed from the CoreNLP server's response. Once all triplets are processed, we execute the queries to create the nodes and relationships of the graph. Thus, we only iterate through the triplets once. From this experience, we realized we should have considered creating an efficient algorithm initially so we would not have to refactor and optimize the code later on.

## 13.2 Future Work (Future Semesters)

There are still more ideas and features that can be implemented to further streamline usability, improve efficiency, and provide more accessibility for our application. We also have some known bugs

### 13.2.1 User Interface

1. Currently, our graph does not support multiple edges between the same two nodes as this would cause edges and edge labels to overlap.

2. For collections that create large graphs, there is a lag between when the user clicks the collection and when the displayed graph changes to the new graph. A loading screen needs to be added over this lag, but we only partially locate where this lag was coming from.

3. When selecting the the info button to display collection info, it would be nice to have an additional modal to display file names that are in the collection as opposed to just numbers of the files. We chose to exclude the names from the info modal because names could often be very long and nondescriptive which cluttered the info modal.

### 13.2.2 Update Collection

The functionality to update collections should be supported in the future. This would consist of adding or deleting documents to existing collections and regenerating a knowledge graph. To help support this feature, we have implemented a file server locally so that the web scraping step does not have to be repeated if the collection was originally made through inputting a text file of URLs.

### 13.2.3 Update User Information

The functionality to update user information should also be supported in the future. Currently, when logged in, the user is only able to see their username and role. To provide more information to the user, their email along with other information should be displayed. The user should also have the option to make any edits to their information, including but not limited to their name, password, and email.

### 13.2.4 Import and Export Collections

The functionality to import and export collections would be a useful feature to be implemented. Currently, users can only view their collections unless they are an admin. If a user wanted to share a graph with another user, they would have to have access to the original user's account to see their graph. If the other user wanted to see their graph, they could create a new graph with the original documents, but that is not guaranteed to be identical. The solution to this is being able to import and export collections. Collections that have been created will have the option to be exported to some file, and these files can be imported back to create a new collection by the requested user.

### 13.2.5 Triple Extraction

During execution and graph creation, we noticed many of the triples didn't produce what we wanted. At first they were redundant, which we found a fix for. From trial and error we still recognize Stanford CoreNLP to be the best method of extracting triples due to our resource and time constraints. However, ideally there would be a machine learning language model fine tuned for triple extraction. This is what's likely to yield the most meaningful and accurate results for triple extraction.

### 13.2.6 Testing

When we finished our project, we got into writing automated tests for the project. We found testing certain aspects of the frontend via code challenging (such as viewing the graph) as it is something best performed by the human eye and not via code. Additionally, in the backend, we found it to be a pain to test as we had to start the containers for Neo4J, Mongo, and Stanford CoreNLP, and reset these databases in between runs. Finding efficient ways to fully test our project was a challenge we think future developers working on this project will be able to improve upon. We solved most of our testing qualms this semester through creative workarounds, such as in the frontend generating random data for textfields to work around the restrictions of Playwright.

### 13.3 Timeline

| | |
|---|---|
| 02/05 | Initial Client Meeting |
| 02/09 | Initialized the Full-Stack Application |
| 02/23 | Created Home Page on Front-end |
| 02/27 | Initial Queries to Neo4j |
| 03/11 | Created Modals for Uploading and Deleting Collections |
| 03/17 | Triplestore Extraction into Graph |
| 03/18 | User Login Creation |
| 03/21 | NER Tagging of Triples |
| 03/23 | Collection Processing and Text Extraction |
| 03/27 | Finalize Backend and Testing |
| 03/30 | Finalize Frontend and Testing |
| 04/15 | Integration, Finalize Project, Testing |
| 05/01 | Finalize Report |

Table 9: Timeline

# 14 Acknowledgments

We would like to thank our client, Dr. Mohamed Farag, for his cooperation and support during the course of this project. We also extend our appreciation to Dr. Farag for his guidance and expertise as our professor, which was been essential to our success in the semester we spent developing this project.

Client Name: Dr. Mohamed Farag
Client Email: mmagdy@vt.edu

# 15 References


[1] “Neo4j graph database & analytics – the leader in graph databases,” Graph Database & Analytics, https://neo4j.com/.

[2] “Performance recommendations - neo4j python driver manual,” Neo4j Graph Data Platform, https://neo4j.com/docs/python-manual/current/performance/.

[3] “Introduction - cypher manual,” Neo4j Graph Data Platform, https://neo4j.com/docs/cypher-manual/current/introduction/.

[4] “Sigma.js,” www.sigmajs.org. https://www.sigmajs.org/

[5] “React,” https://react.dev

[6] “MongoDB Atlas,” https://www.mongodb.com/atlas/database

[7] “bcrypt,” https://pypi.org/project/bcrypt/

[8] “Material-UI,” https://mui.com/material-ui/

[9] ”Stanford CoreNLP OpenIE,” https://stanfordnlp.github.io/CoreNLP/openie.html