Knowledge Graph

Final Report

CS 4624: Multimedia, Hypertext, and Information Access
Virginia Tech
Blacksburg, VA 24061

Client: Prashant Chandrasekar
Professor: Dr. Edward A. Fox
Authors: Daniel Olsen, Daniel Weedon

May 10, 2021

Contents

1 List of Figures **2**

2 List of Tables **2**

3 Executive Summary **3**

4 Introduction **4**
- 4.1 Objective 4
- 4.2 Client Background 4

5 Requirements **5**

6 Design **6**

7 Implementation **7**
- 7.1 React Visualization Demo 8

8 Developer Manual **9**
- 8.1 Installing Grakn 9
- 8.2 Installing HyperNetX 9
- 8.3 Graph Schema 9
- 8.4 Python API 12
  - 8.4.1 API Development 12
  - 8.4.2 API Utilization 12
- 8.5 Methodology 15
  - 8.5.1 User Goals 15
  - 8.5.2 Tasks 16
  - 8.5.3 Implementation-Based Service 17
  - 8.5.4 Workflows 18

9 User Manual **21**

10 Lessons Learned **25**

11 Future Work **25**

12 Acknowledgements **26**

13 References **27**

1 List of Figures

List of Figures

1 Example Knowledge Graph 6
2 React Graph Vis Demo 8
3 Knowledge Graph Schema 11
4 Instance of Rule 12
5 Service Registration Workflow 19
6 Knowledge Graph Database Goal 19
7 API Build Goal 20
8 Postman Workspace 23
9 Postman Header Tab 23
10 Postman Search Request Body 24
11 Postman Search Request Response 24

2 List of Tables

List of Tables

1 Registration Service Table 17

3 Executive Summary

The knowledge graph project is a two-component project. The first component is concerned with the back-end, Grakn.AI, while the second component deals with the front-end service registration. The goal of the project is to build a knowledge graph that represents how user information goals are connected to one another, and ultimately to user tasks and system services. The knowledge graph is connected to a workflow management system for system operation involving the services. Developers register the services and add them to the knowledge graph.

A knowledge graph is a directed graph data representation that stores interlinked entities. Storing data into the knowledge graph allows you to see how this data is connected with other entities in the graph as well as how those are connected. Through this we see the power of a fully fleshed-out knowledge graph. A user may wish to complete a task but has no knowledge about how to complete this task or the tools needed to do so. They can use the knowledge graph and query for this task and thus retrieve the workflow necessary to perform this task including the input files, output files, libraries, functions, and environments.

For this project the deliverables are as follows; the Grakn Knowledge Graph, the Python API, and the front-end visualization. These were built and placed into a remote server hosted by Virginia Tech. Future work is needed on the server to connect the API to the front-end visualization as well as open a port on the server so that visualization is accessible.

4 Introduction

The following section will give an overview of why this project was created and the main objective of the final product. Following this will be a description of the client who oversaw the project and all the work done.

4.1 Objective

This project is a continuation of the previous year's efforts in the project *Twitter-Based Knowledge Graph for Researchers* [1], now generalized for wider reach. A goal for the knowledge graph software is to be general enough to be applicable for different areas so developers of different projects are able to register services into the knowledge graph. The main objective is to create a program that a user can use to query for workflows within the knowledge graph based on tasks they want to get done and retrieve all the information regarding such tasks. The user can also interact with this knowledge graph to add and remove services and tasks from it to expand on the existing information in the database

A knowledge graph is a powerful type of database which is used to connect entities together via hyper-edges. A hyper-edge is an edge which can connect 2 or more entities together. Through this, we can find connections between different entities based on multiple different categories which allow us to visualize workflows and how they are connected internally as well as how they are connected to other workflows.

4.2 Client Background

Our client, Prashant Chandrasekar, is a Ph.D. student working in the Digital Library Research Laboratory at Virginia Tech. This project is a proof of concept for a part of his dissertation.

5 Requirements

In order to be able to run this project locally, you will need to have Java 8 installed which is a prerequisite for Grakn. Next, Grakn can be installed via the guide on their website [2]. Grakn will need to be installed manually because we are using version 1.8.4 and not the latest version. Once Grakn is installed and can be run properly, we must next make sure we have Python. The API was built using Python 3.7.9 and any newer version will not be compatible with Grakn. [3]

In order to run the Knowledge Graph visualization locally, you must be using Python 3.6 or above to run the HyperNetX package. HyperNetX can be installed with the instructions available on their website [4].

6 Design

To understand more clearly what a knowledge graph is, we can see an example in Figure 1. We see that files are connected to each other via tasks and these generate a workflow. In knowledge graphs, we can have hyper-edges; these are edges that can connect multiple entities at once via just that single edge. This allows us to be able to connect multiple files together using just one edge. For example, in the task "Preparation", we see two input files, in the knowledge graph, these would be connected as inputs to the task via one edge instead of two independent edges. Our front-end visualization represents this concept of a hyper-edge by wrapping all the files that are connected via one hyper-edge together.

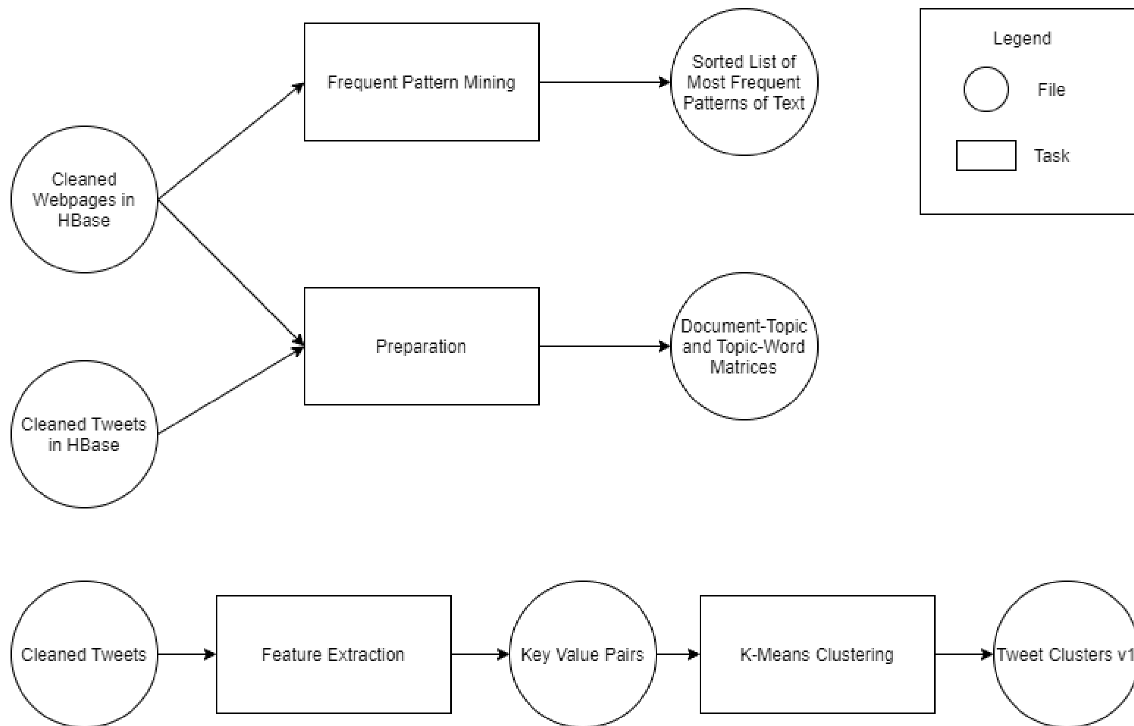

Figure 1: Example Knowledge Graph

7 Implementation

Given the novelty of this project, much time was put into research and learning to determine which framework would best suit us, which library can be used to represent hyper-edges, as well as learning Grakn [2] and how it works. The project was also split in two so each team member could focus on separate independent sections to work on. One of these sections was the back-end implementation of the database in Grakn as well as the Python API. The other section was the front-end client side that users will be able to interact with and see the knowledge graph visualized.

On the back-end, after the initial stages of learning Grakn, the schema was developed for how entities and their connections would be represented in the knowledge graph. (More about this in Section 6.2: Graph Schema.) Once the schema was developed, part of the Python API was implemented to be able to insert our test data into the database which was in CSV format as two separate files: "Nodes.csv", and "Edges.csv". After verification of the data being inserted properly, we then worked on determining the correct queries to access the graph. In parallel to this, whenever a query was finalized such as the query to find a file based on its name attribute, this was then implemented into the Python API and tested. After all the correct queries were discovered and laid out properly and the Python API implemented such, the first half of the API was complete. The final step for the back-end component was to connect this API to the front-end using Flask.

On the front-end, the first goal was to read and review the prior team's work from the previous semester. After that, the next goal was to implement the knowledge graph visualization such that it could be used by the developers to view the state of the graph and ascertain the structure of the knowledge graph. This portion of the project took a large amount of time due to research, trial and error, and attempting to find a library that supports hyper-graphs. During this time, a React demo was developed as a proof of concept for the visualization, but it did not meet our requirements due to the library only supporting one-to-one edges and not hyper-edges. After more researching, the team decided to use a grouping type visualization of a hyper-edge, as *Viral Genomes* project team had already made a knowledge graph visualization that represented our original plans. [5] So, the team decided to choose the grouping method of visualization so that we could compare and contrast the two approaches. Our group-based visualization was implemented using the Python HyperNetX library. [4] The Developer Manual contains more information on how to get started with HyperNetX as well as the other software used in this project.

7.1 React Visualization Demo

The first visualization method we used was a React component called *React Graph Vis*. This React package can be used to display interactive network graphs. The Github page (`https://github.com/crubier/react-graph-vis`) has a demo of the React package. [6] Our visualization used the source code of the demo with slight alternations. The resulting graph had many beneficial features: it was dynamic, movable, easy to edit graphs, and it displayed more information while hovering over a node. Figure 2 shows the resulting graph from the React Demo. The graphs that are produced by this React package are very customizable in their appearance and function, but did not meet our specific needs for this project. Despite its many nice features, this React package did not support hyper-edges in its graphs. The edges could only go from one node to another, so it lacked an important feature which our visualization required.

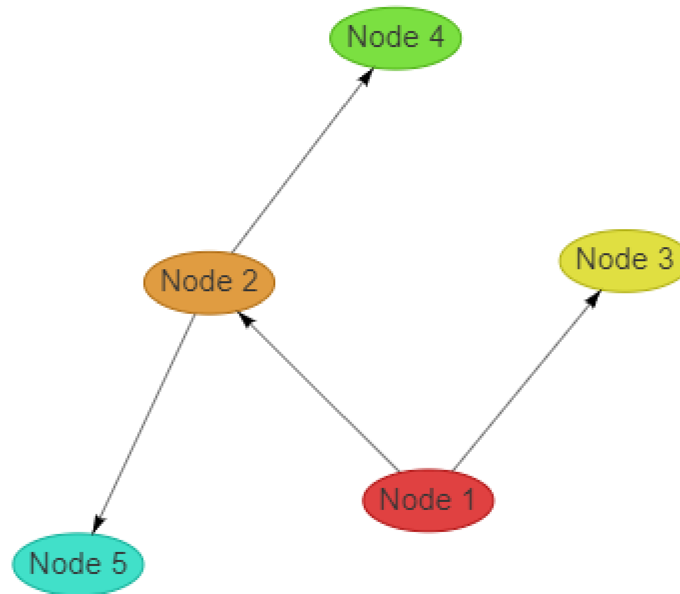

Figure 2: React Graph Vis Demo

8 Developer Manual

8.1 Installing Grakn

For this project, we are using the open source version of Grakn that can be found at the Grakn.ai website at: `https://Grakn.ai/download#core`
During development, Grakn Labs pushed out several new versions of Grakn that would have required substantial modifications of the project in order to keep up. Rather than adjusting to each new version, we developed the project under one older version of Grakn, version 1.8.4. Newer versions such as anything after Grakn 2.0.0 will be incompatible because they changed much of the commands to interact with the knowledge graph.
In addition to this, Grakn Workbase (version 1.3.6), is recommended as well for graph visualization. Newer versions of Grakn Workbase, such as Grakn Workbase 2.0.0, were built to run with Grakn 2.0.0 and may not be compatible. This tool was very helpful at the beginning of development to test out different schemas and datasets. Once Grakn is installed properly, in the terminal we can start the server using:

grakn server start

Once the server is started, we can now create our knowledge graphs and interact with them.

8.2 Installing HyperNetX

HyperNetX is a Python library that is used to visualize the Knowledge Graph. HyperNetX has a website with installation instructions, documentation, and other information available at the following URL: `https://pnnl.github.io/HyperNetX/build/install.html`. [4] For a smooth installation and convenient work space, it is recommended to use an Anaconda environment during installation.

8.3 Graph Schema

Given that the schema is the backbone of the whole knowledge graph, this was thoroughly thought out and optimized to minimize complexity and space where possible. The schema is shown in Figure 3. We see that only 1 entity is defined which is file. The file entity represents the files and their associated information stored in their attributes which are denoted by the keyword "has". In Grakn, we can assign entities different roles that they can play denoted by the keyword

"plays". For example, the file entity plays 4 different roles, 2 of which being the "inputfile" and "outputfile" which allow them to be connected via the task relation. The "task" is a relation which stores its own separate attributes and connects multiple file entities together. One of Grakn's key features is its ability to be able to represent hyper-edges and this is utilized by the fact that we can have a task relation that can relate multiple "input" files to multiple "output" files via only this single relation.

The next type of relation we see is the "instanceOf" relation. This relation is used to connect files together that are instances of another. An example of these connections can be seen in the knowledge graph in Figure 1. This relation connects files or tasks that are instances of other files or tasks, respectively, which allows us to see how different workflows are related to one another.

For the "instanceOf" relation, this needs to be applicable for instances of instances. For example if C is an instance of B and B is an instance of A, C is therefore an instance of A and we need to make sure our graph reflects this properly. To do this, we can create a rule in our schema as seen in Figure 4.

Through this schema we can represent all the files and tasks as well as their connections to one another using only one entity type, two relation types, 1 rule, and any needed attributes for each.

```
define

file sub entity,
    has name,
    has fileId,
    plays inputfile,
    plays outputfile,
    plays parent,
    plays child;

task sub relation,
    has name,
    has taskId,
    relates inputfile,
    relates outputfile,
    plays parent,
    plays child;

instanceOf sub relation,
    relates parent,
    relates child;


name sub attribute,
    value string;

fileId sub attribute,
    value long;

taskId sub attribute,
    value long;
```

Figure 3: Knowledge Graph Schema

```
35   # Instance relation rule:
36   # If A is parent of B
37   # and B is parent of C
38   # then A is parent of C
39
40   instanceOfRule sub rule,
41   when {
42       (parent: $x, child: $y) isa instanceOf;
43       (parent: $y, child: $z) isa instanceOf;
44   },
45   then {
46       (parent: $x, child: $z) isa instanceOf;
47   };
```

Figure 4: Instance of Rule

8.4 Python API

8.4.1 API Development

This section will outline the important steps necessary to get the Python API working on your machine. This is important for continuation of this project with editing the API or adding to it. The API is made using Flask 1.1.2 [7].

Due to compatibility limitations with Grakn, the newest Python version that is compatible with Grakn is Python 3.7.9, which is the version we are using. We need to also make sure that we install grakn-client in Python using the command:

pip install grakn-client

Once we have completed this and made sure Grakn was installed, in our Python client, we need to import Grakn using the following import:

from grakn.client import GraknClient

8.4.2 API Utilization

This section outlines how a client can use the Python API to communicate to the Grakn Database. Followed are the endpoints that are implemented, the

requests that can be made on those endpoints, as well as how to make those requests. The API takes in JSON objects as input and returns outputs in the form of JSON objects.

/File The /File endpoint is used to be able to create and remove Files from the knowledge graph based on given parameters.

POST The POST method is used to create a new file and add it to the graph.

```
{
    "Id" : "exampleId",
    "name" : "exampleName"
}
```

DELETE The DELETE method is used to delete a file in the graph that matches the parameters given.
The IdOperator can take the following forms: $=, >, <, \leq, \geq$
The nameOperator can be either "exact", or "contains". "exact" indicates that you are looking for a file with a name that exactly matches the given "name" parameter. "contains" indicates that you are looking for a file with a name that contains the given "name" parameter.

```
{
    "Id" : "exampleId",
    "IdOperator" : "exampleIdOperator",
    "name" : "exampleName",
    "nameOperator" : "exampleNameOperator"
}
```

/Task The /Task endpoint is used to be able to create and remove Tasks from the knowledge graph based on given parameters.

POST The POST method is used to add a task to the graph based on the given parameters. The input and output both must be given as the fileId of the file you want to be the input or output.

```
{
    "Id" : "exampleId",
```

```
    "name" : "exampleName",
    "input" : "exampleInput",
    "output" : "exampleOutput"
}
```

DELETE The DELETE method is used to remove a task in the graph that matches the parameters given.
The IdOperator can take the following forms: $=, >, <, \leq, \geq$
The nameOperator can be either "exact", or "contains". "exact" indicates that you are looking for a task with a name that exactly matches the given "name" parameter. "contains" indicates that you are looking for a task with a name that contains the given "name" parameter.

```
{
    "Id" : "exampleId",
    "IdOperator" : "exampleIdOperator",
    "name" : "exampleName",
    "nameOperator" : "exampleNameOperator"
}
```

/Search The /Search endpoint is used to be able to search the knowledge graph for files, tasks, or both based on the given parameters.

GET The GET method is used to find one or multiple results that match the given parameters.
The type parameter is used to distinguish what you want to search for and can be any of the following: "both", "file", "task"
The IdOperator can take the following forms: $=, >, <, \leq, \geq$
The nameOperator can be either "exact", or "contains". "exact" indicates that you are looking for a file(s)/task(s) with a name that exactly matches the given "name" parameter. "contains" indicates that you are looking for file(s)/task(s) with a name that contains the given "name" parameter.
The limit parameter allows you to set a limit for how many results you want to be returned to you if there may be multiple matches.
Returns a JSON object which is a list of JSON objects that each represent a match in the search parameter. If there are none, returns an empty JSON object : { }

```
{
    "type" : "both",
    "Id" : "exampleId",
    "IdOperator" : "exampleIdOperator",
```

```
    "name" : "exampleName",
    "nameOperator" : "exampleNameOperator",
    "limit" : "1"
}
```

/Path The /Path endpoint is used to give the path between two given files in the graph. The path returned is the shortest path between the two files if there are multiple paths but functionality can be expanded to include all possible paths in the future.

GET The GET method is used to return the path between two given files if the files exist and such a path exists.
If the files exist and a path between them is found, the method will return a JSON object which contains a list of JSON objects for each file and task that is traversed to go from the first file to the second file.

```
{
    "Id1" : "exampleId1",
    "name1" : "exampleName1",
    "Id2" : "exampleId2",
    "name2" : "exampleName2"

}
```

8.5 Methodology

8.5.1 User Goals

The knowledge graph was designed for users who wish to be able to query for workflows based on a task they want to get done. On top of this, users that are developers want to be able to register their services into the knowledge graph, so they can see how their services potentially connect to other files and tasks. They will be able to do so by seeing a visualization of the knowledge graph and then submit a form with the necessary information about their service and its container. In a more generalized sense then, the users of the knowledge graph need to be able to access and manipulate the knowledge graph.

8.5.2 Tasks

On the Service Registration side, there was only one user and goal: Developers need to be able to register their services to the knowledge graph. The Service Registration Workflow (Table 1) shows the outline of this task which is broken down into sub-tasks. The developer needs to understand the state of the knowledge graph, which is one reason why the visualization part is the first sub-task. Through the visualization, the developer will also be able to select a location to integrate their service into the knowledge graph. Then, they will initialize adding their service to the graph and determine the information that is required for registration. Once the registration is filled out by the developer, the final sub-task is to submit the form; then the service container will be registered into the knowledge graph.

8.5.3 Implementation-Based Service

Service ID	Service Name	Input File Name(s)	Input File IDs (comma-sep)	Output File Name	Output File ID	Libraries; Functions; Environments	API Endpoint (if applicable)
1A	Developer's Service	'User Service Container'	SC1				
2A	View Knowledge Graph	Nodes.json, Edges.json	N1, E1			HyperNetX	Knowledge Graph Python API
3A	Select Service Location						
4A	Add New Service						
5A	Determine Required Information for Registration						
6A	Submit Registration			Registr-ation.csv	R1		
7A	Register New Service in Knowledge Graph	'User Service Container'	SC1				Knowledge Graph Python API

Table 1: Registration Service Table

8.5.4 Workflows

The workflows are used to lay out the user goals and show the steps that need to be taken, as well as the functionality needed to be implemented, in order to meet those goals. One of the first goals was to build the knowledge graph database. A visualization of this goal is shown in Figure 6. We see that in order to achieve this goal, we need to complete the tasks connected to it. Firstly, we need to set up the Grakn server. From there the back-end team needed to learn how to use Grakn and its accompanying language Grakl. Next the schema needed to be developed, and finally the data could be inserted into the database.

The next goal for the back-end team was to build the Python API. We see this goal in Figure 7 with the connected tasks that needed to be completed in order to achieve this goal. In addition, for each of these tasks, there need to be multiple different services implemented which are seen under each task. For example, in order to complete the "Create queries to access the graph" task, we needed to implemented the 4 services listed under it, which are connected by plus signs. On top of this, the last service listed, "Get entity by search parameter", needed two micro services to be implemented in order to complete that service and these are seen connected via the dotted arrows.

The goal of the front-end team was to allow the user to add a service to the knowledge graph. A visualization of this goal is shown in Figure 5, which corresponds to Table 1. In order to complete this goal, the steps labeled "Service 1A" through "Service 7A" must be followed in sequence.

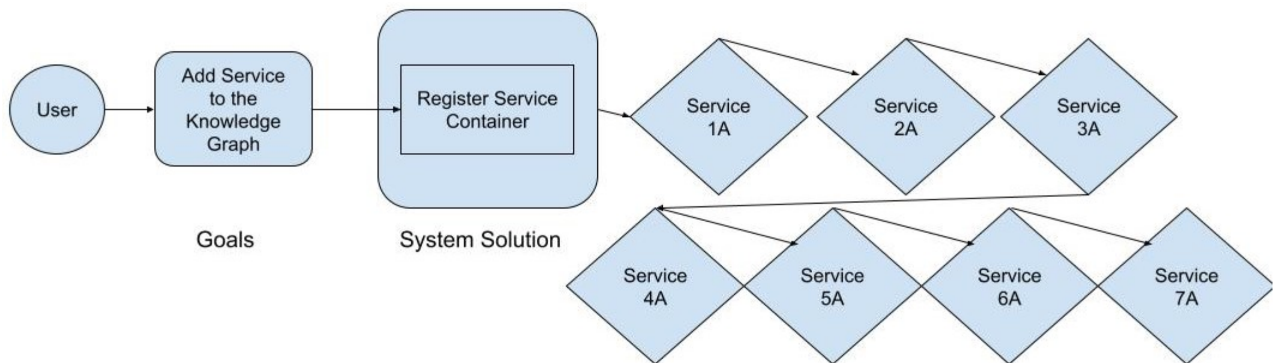

Figure 5: Service Registration Workflow

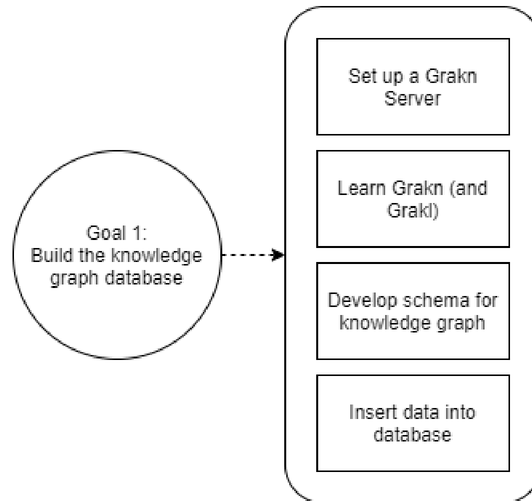

Figure 6: Knowledge Graph Database Goal

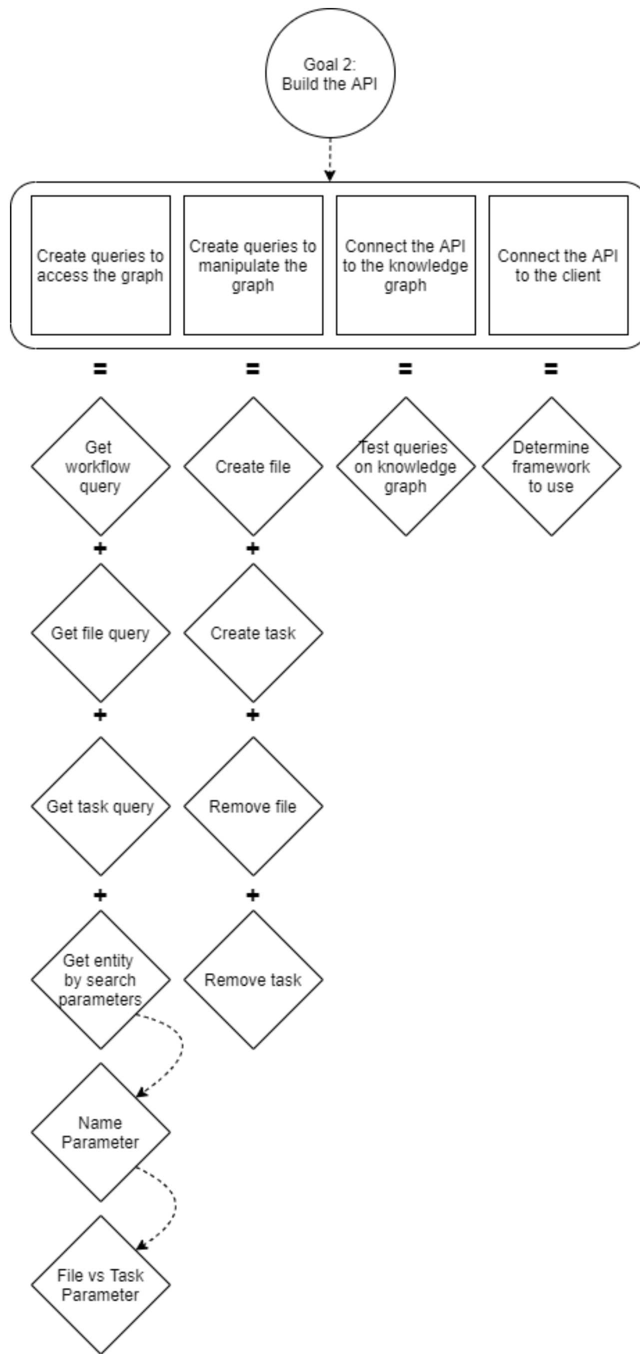

Figure 7: API Build Goal

9 User Manual

The project is on a server but the ports are not publicly open so the deployment is not complete. Regardless, if you have access to the server, you can run the API and use Postman on your local machine to send requests. This will allow you to interact with the knowledge graph and all the data inside it. The server is a Ubuntu server hosted by Virginia Tech, which requires a connect through PulseSecure [8] to access. To be able to log in, you will need to talk to our client Prashant Chandrasekar to get access to the server. Once you can successfully log into the server you will need to perform ssh tunnelling to create a bridge between a port on your local machine and the server. To do this, you can use the command:

ssh -YC -L 10000:localhost:5000 yourUsername@128.173.236.133

This command will ssh you into the server and connect your local machine's port 10000 to the server's port 5000. You may choose any port you wish to use for your local machine, but the Flask API runs on port 5000 on the server. Once logged into the server after this, you will next navigate to

/home/olsendanielv/flask_app

Once in this directory, run the following command:

source venv/bin/activate

This command allows us to run in venv which is needed to be able to use Flask. This can be confirmed with (venv) showing before each prompt. Next return to the previous directory (which can be done with "cd .." and in here you should be able to now enter the directory named "Grakn". In this directly, we have the flaskAPI.py file which runs the API. Before we run this we first need to start the Grakn server. The command to start the server can be run from any directory:

grakn server start

When you receive the success indicators from Grakn we can now run our API with the following command:

python flaskAPI.py

The API is now running successfully on port 5000 and we can send requests to it. To send requests, we will be using Postman which is a platform for API development. [9] You will need to download the desktop version and once done, run the application. The website referenced has a guide on installation and how Postman works if any errors occur. Once Postman is running, we can create a new workspace and enter it. The workspace is shown in Figure 8. Once in the workspace, we can click the plus icon to begin a new request. There is a drop down box to select which request to make and the available requests for our API are documented in Section Python API. To perform a request we need to enter a URL which will begin with:

http://localhost:10000

This will be followed by the type of operation you want to make. For example, if you want to make a GET request in /Search, the URL will be:

http://localhost:10000/Search

Next we need to add a header to our request to specify that we are using JSON format. To do this we need to click the header tab and fill out the information shown in Figure 9 and make sure the check-mark on the left is marked. From here, we can then fill in the body of our request under the Body tab in Postman. The body will contain the information regarding the request we are trying to make. For each request this information and the format of the body can also be found in Section Python API. An example of the body of a search request is shown in Figure 10 . In this example we are searching for all files that have an ID equal to 1. We leave the name and nameOperator fields empty because we do not wish to specify a nam. The limit is set to 1 indicating we only want 1 file to be returned if multiple are found. We see at the bottom of Figure 11 what the server response is.

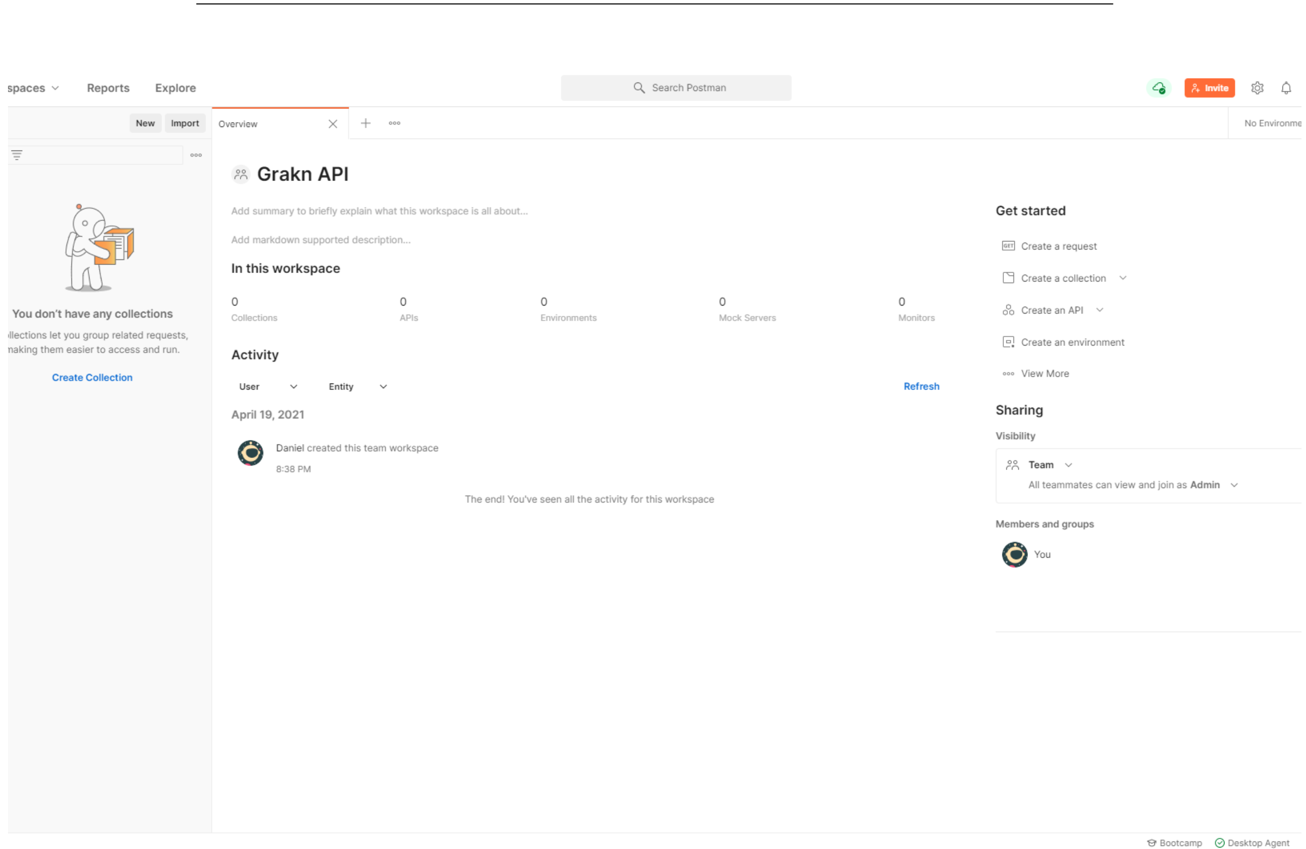

Figure 8: Postman Workspace

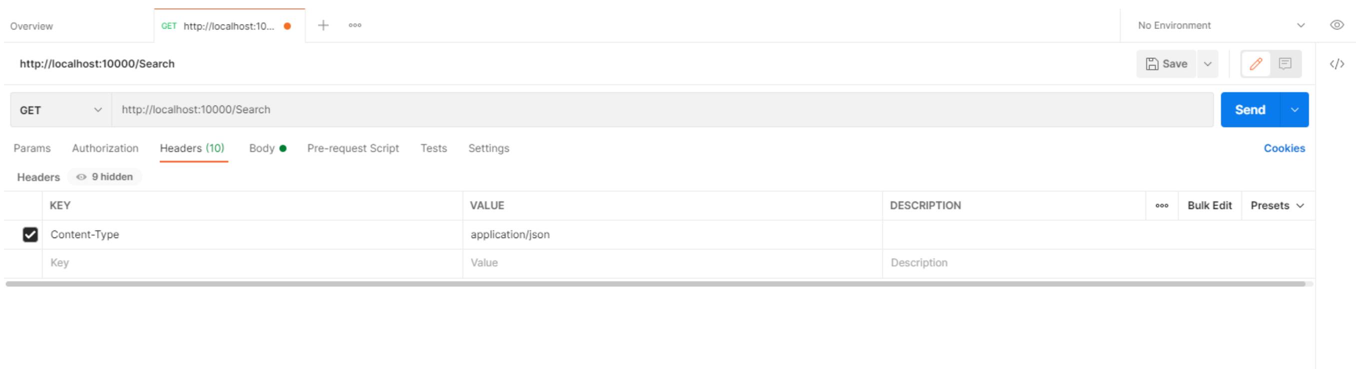

Figure 9: Postman Header Tab

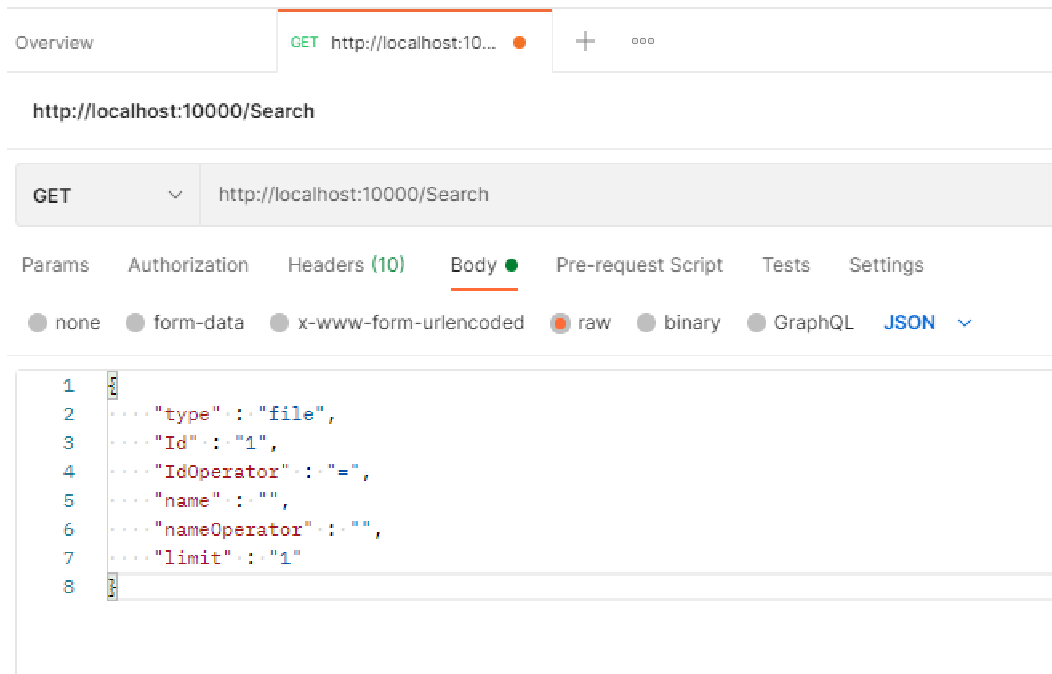

Figure 10: Postman Search Request Body

Figure 11: Postman Search Request Response

The Knowledge Graph visualization is contained in a Python file and requires a .csv file for the nodes and edges to display. The fields that are required in the node .csv file are: field and name. The fields that are required in the edge .csv file are: taskId, name, inputId, and outputId.

10 Lessons Learned

This project was a unique project in multiple aspects. One of the main ones is in the technology. Grakn is new tool that does not have much spotlight in the tech industry which makes learning resources sparse and compatible tools, such as the front-end hyper-graph library, few and far between. This created a large upfront barrier to initial progress on the project because so much time needed to be dedicated to learning and research.
From this, one of the main things our team learned was that when dealing with new technologies, it is important to stay motivated because progress is hardly seen in the beginning stages. Also, making sure to dedicate time for learning/research is crucial with new technologies and was critical for this project in particular.

11 Future Work

Further development of this project is needed to be able to fully implement user functionality. The goal is to connect the front-end and back-end together so that the user can interact with the front-end which makes these requests to the back-end. Achieving this requires an open port as well as communication between the different applications. The visualization has only been run locally and may need modification in order to work once a port is open on the server. In addition, future work is needed to update Grakn to the latest version, currently 2.0.2, which made major changes to how database interactions are performed. The API will need to be modified concurrently as well to be compatible with the version of Grakn that will be used. Furthermore, a registration form needs to be developed and integrated into the front end so that developers can register services into the Knowledge Graph. This registration form would need to make calls to the Python API to update the Knowledge Graph in the database.

NOTE: There is a bug with the Python decorator package version 5.0.0-5.0.4 and 5.0.6 which causes a NetworkX error in random_state. The working decorator versions are: before 5.0.0, 5.0.5, or 5.0.7. [10]

12 Acknowledgements

We would like to acknowledge our client and advisor Prashant Chandrasekar as well as our professor Dr. Edward A. Fox for their guidance throughout the project. We would also like to thank the groups of students from prior semesters whose work we built upon during this project.

13 References

References

[1] E. Meno and K. Vincent, "Twitter-based knowledge graph for researchers," Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, Tech. Rep., 2020. [Online]. Available: http://hdl.handle.net/10919/98239 (visited on 05/09/2021).

[2] G. Labs, *Grakn homepage*, 2020. [Online]. Available: https://grakn.ai/ (visited on 05/09/2021).

[3] G. van Rossum, *Python homepage*, 2021. [Online]. Available: https://www.python.org/ (visited on 05/09/2021).

[4] B. Praggastis, *HyperNetX (HNX)*, 2018. [Online]. Available: https://pnnl.github.io/HyperNetX/build/index.html (visited on 05/09/2021).

[5] M. Ambrose, C. Turner, D. Pompeii, J. Fisher, and J. Hubert, *2021project-viralgenomes*, Feb. 2021. [Online]. Available: https://canvas.vt.edu/courses/125164/pages/2021project-viralgenomes.

[6] V. Lecrubier, *Crubier/react-graph-vis*, Mar. 2021. [Online]. Available: https://github.com/crubier/react-graph-vis.

[7] A. Ronacher, *Flask homepage*, 2010. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/ (visited on 05/09/2021).

[8] 4Help Online, *Pulsesecure VT guide*, 2021. [Online]. Available: https://www.nis.vt.edu/ServicePortfolio/Network/RemoteAccess-VPN.html (visited on 05/09/2021).

[9] Postman Inc, *Postman homepage*, 2021. [Online]. Available: https://www.postman.com/ (visited on 05/09/2021).

[10] D. Schult, *Error in 'random_state' decorator*, Apr. 2021. [Online]. Available: https://github.com/networkx/networkx/issues/4718 (visited on 05/09/2021).