



ETH AI CENTER



United  
Nations

Practical Work - Project Report

# SemUN: A Semantics-Powered Search Platform for the United Nations' Digital Library

Clément Sicard  
csicard@ethz.ch  
D-INFK, ETH Zurich

*Supervised by:*

Dr. Menna El-Assady<sup>1</sup>, Dr. Sascha Langenbach<sup>1</sup>, Catherine Pysden, MSc.<sup>2</sup>

<sup>1</sup>ETH Zurich <sup>2</sup>United Nations

August 30, 2023

## Keywords:

*Natural Language Processing (NLP), Named Entity Recognition (NER), Graph databases,  
Frontend, Network Visualization*

## 1 Introduction

The United Nations Digital Library (UNDL) is a United Nations (UN) service that provides public access to a diverse range of UN documents: voting data, speeches, maps, and open access publications starting from 1979. All of these documents have been classified according to the UNBIS Thesaurus, a multilingual database of the controlled vocabulary used to describe UN documents and other materials in the Library's collection, with a more or less precise topic label.

The main idea of this project is to create an analysis platform, with a network visualization of documents from a subset of the documents in the UN Digital Library. The analytics platform includes a basic Named Entity Recognition (NER) system to extract mentioned entities from the documents. The visualization part will be a network visualization, leveraging the versatility of the structure of a graph to display the extracted insights. Both parts aim at improving the search of documents by implementing an analytics layer on top of the existing search engine from the digital library.

## 2 Motivation & Scope

In the short-term, the goal of this project is to provide an MVP of a potential future UN product, in close contact with UN staff to make it conform to their needs. Specifically, this project will focus – as a first iteration – on Catherine Pysden's suggestion, "Women in Peacekeeping".

The project's long-term scope is to be used as a search engine for UN staff, member-state delegates, and members of the public with an interest in UN topics. Potential future work could include extending the project to the whole UN Digital Library. It could, for instance, also suggest Thesaurus-compliant metadata for untagged documents to facilitate the work of Library staff.

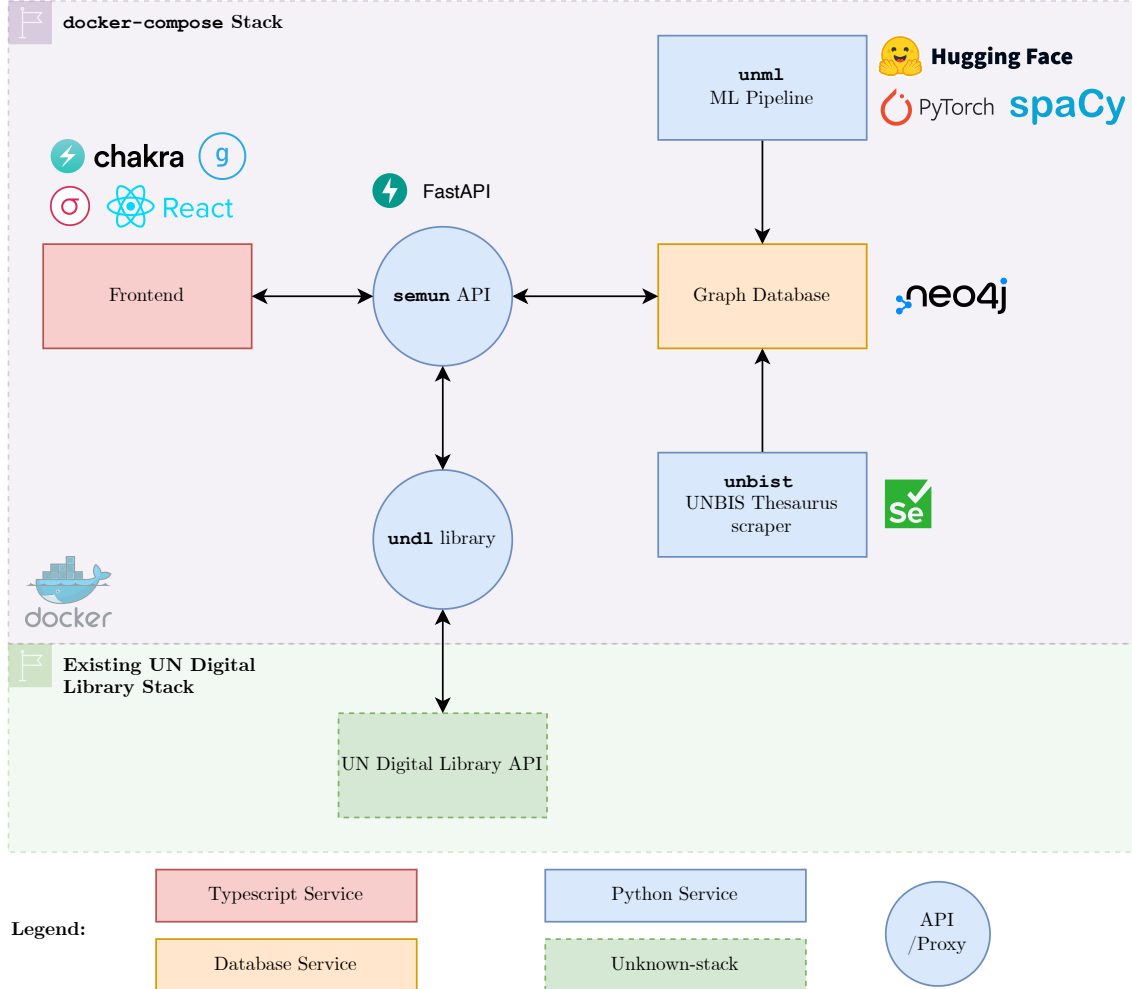


Figure 1: Final stack architecture

### 3 Final architecture

The final architecture is a full-stack architecture, from the database to the frontend, handed in as a Docker compose stack: **un-semun**.

The project consists of 7 GitHub repos:

- **un-semun:** The main repository with the **docker-compose** stack declaration.
- **un-semun-frontend:** The frontend.
- **un-semun-api:** The API for the frontend.
- **undl:** The code for undl, a Python wrapper around the UN Digital Library API.
- **un-unbis-thesaurus-scraper:** A scraper for the UNBIS Thesaurus taxonomy website.



### 3.2 un-semun-api: An API for un-semun-frontend using FastAPI

The **un-semun-api** Python package was developed to provide an API for the frontend. It is a **FastAPI** application, which is a Python framework for building APIs. It is a modern framework, which is fast, easy to use, and well documented. One nice feature is that it is also coupled with **pydantic** to perform data validation and serialization. This is very useful to ensure that the data sent to the frontend is valid, and to avoid having to write boilerplate code for serialization and deserialization.

This package mainly acts as a proxy between the frontend and the UNDL API, and is composed of 2 main endpoints:





- **/search**: This endpoint is used to perform a search query. It takes a **GET** HTTP request with query parameter the prompt. Then, it uses **undl** (3.3) library to perform the search on the UNDL API and returns the results to the frontend.
- **/graph**: This endpoint is used to get the graph corresponding to the above search query, also with an HTTP **GET** request with the prompt as unique query parameter. To optimize the process, it first gets the IDs of all documents returned by the search query on the given prompt, then it queries the Neo4j graph database (3.6) to get the graph corresponding to these documents. Finally, it returns the graph to the frontend as **JSON** structured according to the format expected by the **graphology** Typescript library, which is used to model a graph and manipulate it as a programmatic object.

Note that the API is dockerized and that it offers a basic query caching mechanism to avoid querying the UNDL API too often. This is done using a simple **dict** in memory, which is not ideal but enough for the scope of this project.

### 3.3 undl: A Python library to wrap to the UN Digital Library API

For the **undl** library, I used Python 3.10 with packages, with **requests** for the HTTP requests, **pandas** for the data manipulation, and **pydantic** for the data validation.

Its purpose is to query the UN Digital Library API, and to return the results in a structured way and convenient way. It converts the **MARCXML** response from the API to a **JSON** object and implements caching to save the results of the queries, and to avoid querying the API again if the same query is made. The client offers these main methods:

- **query(...)**: Takes the prompt (e.g., "Women in peacekeeping") as an argument, and returns the detailed results of the query as a **JSON** object. The API response is a list of detailed documents (the fields we actually collect are precised in 3.6 for the **Document** node type). The main difference with a direct call to the UNDL API is that it converts the output to a **JSON** object. It is used by  **un-semun-api** (3.2).
- **getAllRecordIds(...)**: Same as **query**, but returns only the IDs of the documents, not their entire fields - hence is much faster. It is used by  **un-semun-api** (3.2) and  **un-ml-pipeline** (3.5).
- **queryBdId(...)**: Queries the API for a single document, given its **id**. It is used by  **un-ml-pipeline** (3.5).

These methods are callable either from the command-line interface (CLI), or from another Python script when imported as a library (see an example in Figure 3)

Note that **undl** requires a valid UNDL API key to work (set using the environment variable **UN\_API**), because the API calls are authenticated by this 36 characters long key.



Figure 3: Querying the UNDL API from a Python script using `undl` library

### 3.4 `un-unbis-thesaurus-scraper`: the UNBIS Thesaurus scraper

🔗 `un-unbis-thesaurus-scraper` is a Python scraper that crawls the UNBIS Thesaurus website<sup>1</sup> and extracts the thesaurus terms and their relations. Once all the Thesaurus entries have been parsed, they are inserted with all their fields to the graph database instance (3.6).

It is mainly used to link Thesaurus topics the documents which are passed through the machine learning pipeline (3.5), but I felt it was also interesting to be able to visualize all the topics together and how they're related, so I built and deployed a small Vercel app to do so: ☁️ <https://un-graph-ui.vercel.app/>, which is built on a similar stack as the frontend (3.1).

### 3.5 `un-ml-pipeline`: The machine learning pipeline

🔗 `un-ml-pipeline` is the main component of this project. It englobes a Python library, `unml`, as well as an API wrapping it. It functions as follows (see Figure 4):

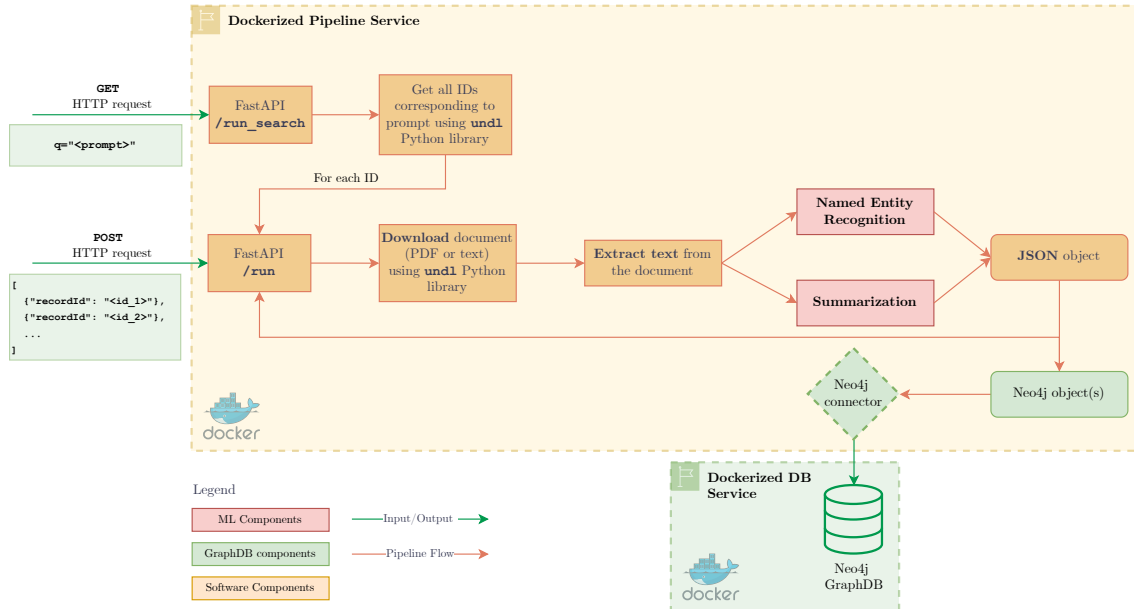


Figure 4: `unml` library: a dockerized machine learning pipeline for NER & summarization

The pipeline API is a `FastAPI` app, which exposes two endpoints. The `run` endpoint gets an ID as input, and works as follows:

<sup>1</sup><https://research.un.org/en/thesaurus>

1. A client sends an HTTP `POST` request to the pipeline API on `run` endpoint, with the payload containing a list of JSON objects with a `recordId`. (Note: this is arguably not an optimal data structure to store the IDs, but it was the easiest way to make `pydantic` happy and validate the input data structure.)
2. Use `undl`'s client `queryBdId` method to download the corresponding document complete information and PDF document. Sometimes it doesn't exist – if it is the case, create a JSON with solely the information from the UNDL API, and jump to step 6
3. Extract the text from the PDF document using `PyMuPDF`<sup>2</sup> library.
4. Then the pipeline is split into 2 parts:
  - **Named Entity Recognition:** Extract named entities from the document. All types entities are collected (the types can vary depending on the model used though), but for data quality reasons, only UN bodies and Countries are actually stored in the graph database and linked to the document. However, the other entities are still returned by the API at the end of a pipeline run, and can be used for further analysis.
  - **Summarization:** The document is being summarized using a deep learning model, and the chosen model is a parameter of the pipeline (see Table 1 for more details). The summary for a document is then stored in the database to enhance the nodes.
5. The results of the machine learning components (e.g., the summary and the extracted named entities) are then stored into a JSON object
6. The JSON object is converted to a Cypher (the graph database, Neo4j, query language) query and then the query is run against the Neo4j database to update the graph. Note that the Neo4j instance lives in another Docker container to better compartmentalize the stack.

The machine learning pipeline also offers the `run_search` endpoint, which receives a prompt as input (e.g., "Peacekeeping"), and works as follows:

1. A client sends an HTTP `GET` request to the pipeline API on `run_search` endpoint, with the prompt as a query parameter `q`.
2. The pipeline API queries UNDL API against the given prompt and collects document IDs corresponding to the search results using `getAllRecordIds` method from `undl`'s. It then returns the list of IDs.
3. For each record ID, do what querying `run` endpoint does

### 3.5.1 Models

When it came to choosing a model for both tasks, the main criteria were the following:

- The model needs to be accurate
- The model needs to have fast inference due to the large number of documents the pipeline could have to deal with.
- The model should ideally have optimized inference for CPU, as the pipeline would preferably not be GPU-accelerated for cost reasons.

---

<sup>2</sup><https://github.com/pymupdf/PyMuPDF>

Hence, I decided to use the `transformers` library by HuggingFace, coupled with their SafeTensors<sup>3</sup> technology for faster loading and `accelerate`<sup>4</sup> Python library for faster inference on CPU. Some models also offered ONNX runtime binaries, but not all of them, so I didn't bother using them – this might however be a good idea for future work and to keep this pipeline cost-efficient.

For summarization, one of big challenges when choosing a model was that it should handle a very number of tokens as input. Indeed, the self-attention layer in the transformer architecture scales quadratically with the input length. Hence, transformers are often non-suited for very large inputs – and reports from the UNDL are sometimes very long and counts 100k+ tokens. I came up with these two solutions to address this issue:

- **Divide & Conquer approach:** Recursively split and summarize the document into smaller chunks, then concatenate the summaries and pass them into the model until the length of the result summary is below fixed threshold.
- **Use transformers models that offer a very large input size:** Some models like LongT5 or LongFormer replace the self-attention mechanism with a custom attention mechanism that scales linearly with the input. This allows them to handle very large inputs. However, they are not as accurate as the other models, and they are also slower to train and to run inference on.

Here is a comparison of the models I used for the summarization task:
















	DistilBART-CNN	DistilBART-XSUM	DistilPegasus-CNN	LongFormer ( <i>default</i> )	LongT5
<b>File</b>					
<b>Paper</b>	<a href="#">arXiv</a> 	<a href="#">arXiv</a> 	<a href="#">arXiv</a> 	<a href="#">arXiv</a> 	<a href="#">arXiv</a> 
<b>Authors</b>	Shleifer et al.	Shleifer et al.	Shleifer et al.	Beltagy et al.	Guo et al.
<b>Company</b>	HuggingFace	HuggingFace	Google	Allen AI	Google
<b>Year</b>	2020	2020	2020	2020	2022
<b>HuggingFace</b>	<a href="#">Link</a> 	<a href="#">Link</a> 	<a href="#">Link</a> 	<a href="#">Link</a> 	<a href="#">Link</a> 
<b>Model</b>					
<b>Max Token</b>	1'024	1'024	1'024	16'384	16'384
<b>Input Size</b>					
<b>Params</b>	306M	222M	370M	162M	248M

Table 1: Models used for summarization task

For Named Entity Recognition (NER) tasks, other models than FLERT work slightly better but are much slower due to their much larger number of parameters. In addition, spaCy has optimized inference for CPU, which is a big plus for this project. Here is a comparison of the models I used for the NER task:

<sup>3</sup><https://huggingface.co/docs/safetensors/index>

<sup>4</sup><https://huggingface.co/docs/accelerate/index>









	FLERT ( <i>default S</i> )	RoBERTa	spaCy <code>en_core_web_trf</code>
File	 <a href="#">arXiv</a>	 <a href="#">arXiv</a>	 <a href="#">arXiv</a>
Paper	<a href="#">arXiv</a> 	<a href="#">arXiv</a> 	-
Authors	Akbik et al.	Liu et al.	-
Company	Flair NLP	Meta (fine-tune HuggingFace)	spaCy
Year	2020	2019	2023 (v3.6.1)
HuggingFace Model	<a href="#">Link</a> 	<a href="#">Link</a> 	<a href="#">Link</a> 
Params	20M(S) 560M(L)	355M	355M




Table 2: Models used for NER task

### 3.6 Neo4j graph database

A graph database seemed to be the best choice to store the data extracted from the documents and the links between them, since it captures this relationship property and is able to retrieve it very efficiently. Neo4j seemed to be the most accessible one to use, but other solutions exist (such as ArangoDB<sup>5</sup> or OrientDB<sup>6</sup>).

Here is the chosen data model for this project:

#### 3.6.1 Types of nodes

- **Document:** This node type represents a document in the UNDL. It contains the document’s id from the library management system, its summary (created when the document was run through the machine learning pipeline), symbol (some internal UN classification for the document - for instance A/C.5/43/SR.50), publication date, title, URL, and publication location.
- **Topic:** This node type represents a topic in the sense of the UNBIS Thesaurus taxonomy. I made a scraper  `un-unbis-thesaurus-scraper` (3.4), and used it to retrieve all the topics from the taxonomy website, and inserted them into the database instance. Each topic has `name`, `cluster`, `id` fields, as well as label fields for each of the 6 official UN languages (`labelEn`, `labelFr`, `labelEs`, `labelAr`, `labelZh`, `labelRu`).
- **MetaTopic:** UNBIS Thesaurus is structured as a hierarchy, and a **MetaTopic** is one of the top-level topics in this hierarchy. It has the same fields as a **Topic** node, and there are 18 of them.<sup>7</sup>
- **Country:** This node type represents a country as a member state of the UN. It has the same fields as a **Topic** node, and there are 193 of them.<sup>8</sup> The data to create these nodes was retrieved from the UN website, and the Cypher queries were generated using a Python script in  `un-semun-misc` (3.8).
- **UNBody:** This node type represents a body of the United Nations. A body of the UN is an organizational unit within the United Nations system, established to carry out specific functions ranging from peacekeeping and humanitarian aid to diplomatic negotiations and policy recommendations. Some famous examples include the Security Council, the World Health Organization (WHO), the International Monetary Fund (IMF) etc... As for the countries, the data was retrieved from the UN website, and the Cypher queries were generated using a Python script in  `un-semun-misc` (3.8).

<sup>5</sup><https://www.arangodb.com/>

<sup>6</sup><https://orientdb.org/>

<sup>7</sup>The list of **MetaTopic** nodes from UNBIS Thesaurus website

<sup>8</sup>The list of the 193 UN member states from the UN website



### 3.6.2 Types of relationships

These nodes are linked together by the following relationships:

- `-[REFERENCES]->`: This relationship type links a `Document` node to a `Country` or `Entity` node, and is self-explanatory: the document references the target entity. This relationship is extracted by the machine learning pipeline.
- `-[HAS.SUBTOPIC]->`: Links a `MetaTopic` node to a `Topic` node to explicit the hierarchy between the two nodes. It was created by the scraper.
- `-[IS.ABOUT]->`: Links a `Document` node to a `Topic` node to indicate that the `Document` has been classified by UN Digital Library staff as a document about the `Topic`. This relationship is extracted from the UNDL API response.
- `-[RELATED.TO]->`: Links a `Topic` node to another `Topic` node indicate a semantic link between the two topics. This relationship is extracted from the UNBIS Thesaurus website, using the scraper as well.

### 3.7 un-semun: The main repository

`un-semun` is a repository that englobes all other repositories as submodules. It also contains the `docker-compose` stack declaration, which points to Dockerfiles in submodules. They are updated using the `Makefile` when new commits are added to the submodules. The port forwarding definitions and environment variables are also declared here. This is the main entry point to run the whole stack.

### 3.8 un-semun-misc

This repository contains scripts for the both the UN member states and the UN bodies: it first extracts the data from an Excel file and a CSV file respectively, then cleans it, and finally inserts it into the Neo4j database after having written a Cypher query. The scripts are written in Python and directly use the `neo4j` Python connector library.

### 3.9 General tech stack notes

All the services composing of the stack are dockerized (i.e., they live in their own Docker container but their storage and network interface can be shared), and they are orchestrated and connected using `docker-compose`<sup>9</sup>.

For all the Python components in the stack (the blue components in Figure 1), the dependencies are managed using `poetry`<sup>10</sup>.

## 4 Discussion & future work

Need to run on all the documents, not just on the "Women in Peacekeeping" documents.

Models are kind of limited sometimes, would probably need to retrain them

Someone more experienced in frontend design could make it even better

<sup>9</sup><https://docs.docker.com/compose/>

<sup>10</sup><https://github.com/python-poetry/poetry>

This approach can be seen as a first draft of a more general approach, applicable to the whole UN Digital Library. It therefore contains a few limitations that fell out of scope for this project:

- **No support for multiple languages yet:** The UN Digital Library is available in six languages. The models used in the machine learning pipeline (3.5) currently only supports English on the summarization tasks, due to the lack of good and convenient enough pre-trained models for other languages. This could be the scope of a future work to train models for the other 5 official UN languages. However, English is the most used language in the UN Digital Library and most recent documents are translated in all 6 languages, so this limitation is not a major one.
- **Limited support for different types of named entities:** Currently, only countries and UN bodies are actually ingested into the graph database, but the models also extract other named entities such as persons, organizations, locations, dates, etc. This could be the scope of a future work to ingest these entities into the graph database.
- **No support for the full text search yet:** The full text search is not yet supported by the graph database nor the frontend. This could also be part of a future work.
- **Performance on the ML tasks can be improved:** The models used to perform the ML tasks are state-of-the-art, but we could see an improvement by fine-tuning them on UN Digital Library specific documents.
- **Availability of ML-enhanced documents:** Due to a lack of resources (cost, time and computing power), the machine learning pipeline has not been run on the millions of documents that compose the UN Digital Library. Before deploying such a system in production, it would be necessary to run the pipeline on all the documents and ingest the results into the graph database, so that the frontend can display the documents consistently, on any query, and not only the ones linked to the "Women in peacekeeping" prompt - indeed, documents will only be present in the database if they have been passed through the ML pipeline, which ingests the documents at the end of the pipeline.

## Conclusion

This project offers a full stack draft product for the UN Digital Library, which is a first step towards a more general approach, applicable to the whole UN Digital Library corpus. It contains a frontend, a graph database, an API, a machine learning pipeline to enhance the documents, a UN Digital Library API wrapper packaged as a Python library, a visualization of the UNBIS Thesaurus taxonomy.

Due to the reduced scope of the project, this can be seen as a basis for a larger work, which would include the improvement stated in the Discussion section (4).

The accent has been put on clean and documented code, maintainability, and the stack can easily be deployed, on a large scale, on a Kubernetes cluster since every service that compose it runs in a separate Docker container.

Special thanks to Catherine Pysden, Dr. El-Assady and Dr. Langenbach for their precious help and guidance throughout the project.