System Specification

General Overview

The task of this homework is to create an application, which is capable of visualizing the relationship between characters of a story in the form of a graph. We consider two entities to be related, if they are mentioned in the same sentence.

Development Team

Members

Team member name	Neptun code	Email
Attila Nagy	ER2KCI	attila.nagy234@gmail.com
Christian Conforti	F8R430	conforti.christian@gmail.com

Detailed Task Description

In this section we outline our plan for solving the problem and introduce the high-level architecture for the system.

Users are able to use the application with or without being authenticated. They can submit a text as input and the application aims to visualize the entities and the possible connections between them as a graph. After a text input is successfully submitted (this is determined by a separate validation service), the entities (nodes) and connections (edges) will gradually appear on the screen, as the NLP services detect them. In addition, authenticated users will also have the option to reload previously submitted text inputs and view the graph visualizing it.

Example Workflow

- User with the correct credentials is authenticated.
- Text input is submitted and sent to the validation service.
- Text input is accepted by the validation service (200 OK) and the data is passed on to the backend.
- The backend calls the sentence splitter service that processes the input and splits it into sentences.
- For each sentence the named entity extractor service is called. This service extracts the entities from the sentence and returns them to the backend.
- The backend dispatches an event for every entity and entity-relationship detected.
- The client receives the dispatched events and displays the detected node or edge on the UI, if it is not yet shown.
- After the whole input is processed by the named entity extractor service the complete graph visualization is stored in a document database.

Planned Components

In order to reach the desired visualization, we need a mechanism, which extracts named entities on a sentence-level from a document. Since this requires natural language processing, we constrain the homework to English input text only, although our design would easily be extendable to other languages as well. We plan to implement three individual NLP components as microservices, which we detail below.

sentence-splitter

A RESTful microservice that receives the entire input document as part of a JSON payload. Returns the sentence boundary offsets in a JSON format.

token-enricher

A RESTful microservice that receives a sentence as part of a JSON payload. Returns the list of tokens with some additional enrichments. There are a handful of subcomponents inside this service that we need to implement. The first is an English tokenizer, which splits the sentence to a list of tokens. This is mandatory, because the NER tagger (which will perform the entity recognition) expects a sequence of tokens. The second is a lemmatizer, which is necessary to ensure that the output graph is not noisy. If we use the surface form to identify unique nodes in the graph, the graph could be populated with multiple nodes corresponding to the same entity. For this reason we will use the lemmas to create nodes in the graph. Finally, an optional component is a part-of-speech tagger, which is a useful (sometimes mandatory) feature in lemmatizer models. The service returns a list of tokens with token-level enrichments in the following JSON format:


```
{
"tokens": [
        {
            "text": "hello",
            "type": "TOKEN",
            "startOffset": 0,
            "endOffset": 5,
            "lemma": "hello",
            "universalPos": "X"
        },
        {
            "text": " ",
            "type": "INLINE",
            "startOffset": 5,
            "endOffset": 6
        },
        {
            "text": "world",
            "type": "TOKEN",
            "startOffset": 6,
            "endOffset": 11,
```

```
            "lemma": "world",
            "universalPos": "X"
          }
        ]
}
```


named-entity-extractor

A RESTful microservice that receives a sentence as part of a JSON payload. Returns the new nodes and edges (based on the recognized entities) that the backend can forward as events to the client. To break the sentence into tokens, the named-entity-extractor calls the token-enricher service. The entity recognition will be performed by a deep learning model trained for sequence tagging. Our choice of architecture is a biLSTM model, because it works particularly well for sequence tagging [1], it is comparable in performance to more recent, transformed-based models and they are much more scalable than their newer counterparts. Assuming the entities *A*, *B* and *C* were detected in a sentence, the following response will be provided by the service:


```
{
  "nodes": ["A", "B", "C"],
  "edges": [
    ["A", "B"],
    ["A", "C"],
    ["B", "C"]
  ]
}
```


frontend

The application is going to have a very simple user interface. The main screen will consist of a header with authentication related controls (sign up, sign in, authenticated user name) and two columns. One for the text input and the other for the graph visualization output of the application. If the user is authenticated, the header will contain a navigation link to a page containing the previously visualized texts. The previous visualizations will be displayed in a list. Each list item will have a show visualization button. If the user clicks the button, the graph will be displayed on the screen.

graph-visualizer-service

Part of the frontend. It is responsible to render a graph (nodes and edges) on the screen.

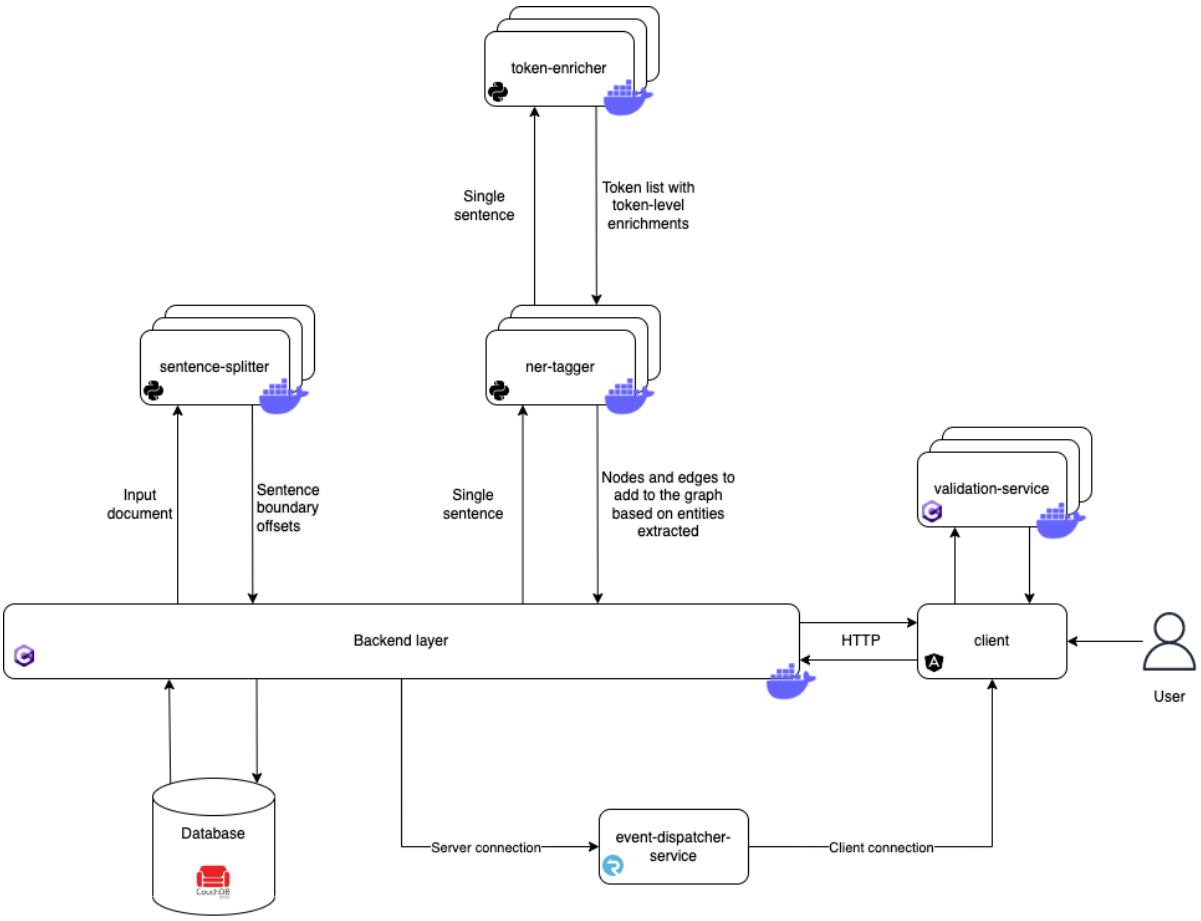
bff (backend-for-frontend)

The backend receives requests from the frontend. Its main responsibility is to call the NLP services asynchronously for every input sentence and continuously dispatch events with the processing results to the client. It is also responsible for managing users and authentication. For authenticated users after the whole input has been processed the result is stored in a document-oriented database like CouchDB.

validation-service

The client calls the validation service when the user submits an input. The responsibility of this service is to make sure that the input provided does not contain any malicious data that could potentially harm the system. If the input is accepted by the service, a success status code is returned to the client.

The complete planned architecture can be seen below:



Planned Tech Stack

We plan to use the following tech stack for implementing each component:

- Typescript and Angular 11 for implementing the frontend.
- ASP.NET 6 and C# 10 for implementing the backend.
- SignalR for sending events from the backend to the frontend.
- For implementing the NLP components:
    - Python 3.9
    - Poetry for dependency management
    - Keras/Tensorflow for the deep learning model
    - Falcon and Gunicorn for the Python service layers.
- CouchDB for the document-oriented database.
- Docker for packaging services, docker-compose for deployment.
- Git for version control.

References

[1] https://arxiv.org/abs/1603.01360