# Project Text Mining API

*Start : October, 2th, 2020*
*End : October, 9th, 2020*
*Zakaria KADDACHE*

# Subject

## Goals

This project aims to develop an API RESTful JSON in Python to perform text mining.
Therefore we would like, from a given text, to extract the people quoted in the text, collect information about them, and retrieve common characteristics.

Input : Text.
Output :
- List of people with information such as type, label, description, image, etc.
- Group by eventual common characteristics such as nationality, age, job, etc.

## Deliveries

- **English log board:** explain the architecture, steps of thinking, improvement for production.
- **Docker image:** with READ-ME containing instructions to test the project.
- **GitHub repository:** source code with clear services (single responsibility) and unit tests.

Bonus :

- API JSON could be encapsulated into a docker container.
- 12 factors development (https://12factor.net/)
- Graphic interface in JS for the user to test the API.
- Propose cool features for the app.

## Tools proposed

Entity extraction: API TextRazor.

Collecting entity information: Wikidata.

# Design & Development

Regarding the given deadline, we would like to build an API quickly. However, we are not sure how this API could evolve in the future. Hence it is important to provide a vision of its possible use in the next version, and to design the API accordingly.

## Functionalities

Let’s clarify the functionalities that our first version should implement, and propose a vision of how it could progress.

The API should be able to receive requests with options in order to perform a text analysis.

### What should the API request look like ?

The request, as a JSON object, should contain:

- the text to be analyzed,
- the options to configure how the analysis should be performed (i.e algorithms and methods to use),
- the options could also define the results the user is interested in (i.e persons with the same job title).
  For example today we are interested in the persons within the text, but tomorrow we may want to retrieve the organisations.

### How the text analysis should be performed ?

We can already identify some steps within the text analysis.

- Entities extraction: weather it is a person, an organisation, or anything else, the first step is to extract entities from the text.
  For now we have been advised to use the *TextRazor API*, but we should take into account that we may also want to implement other methods.
- Entities research: next we need to collect information regarding these entities. For a person it can be the age, the job title, etc.
  Same here, for now we have been advised to use *Wikidata*, but we should take into account that we may also want to implement other methods to collect information (internal databases, etc).
- Entities aggregation: once we have our entities with enriched information (could be coming from different sources and different types) we would have to aggregate this information in order to respond to the given request. (Note we see this aggregation step, as any task we would like to perform on our entities, such as finding relations, clustering, etc.)
- Finally we send back the response of our API, or execute any other task (send email,...)

### What should the API request look like ?

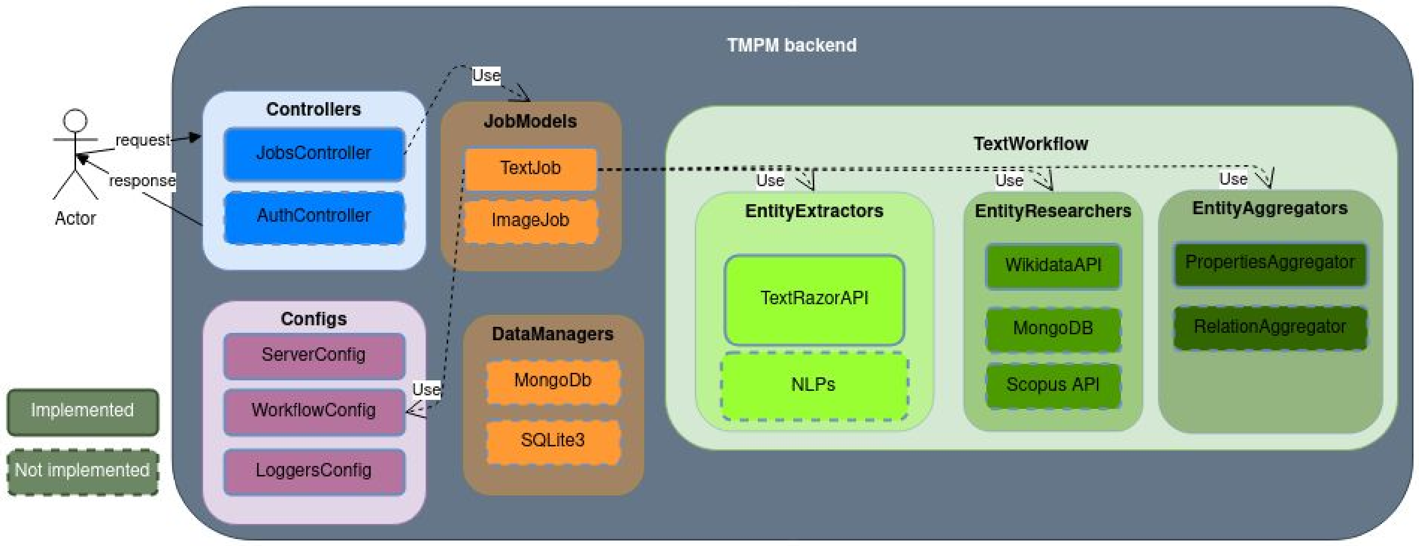
The response of our API, as a JSON object, should contain:

- an array of all entities found in the text, with all the information we could have collected about them. Each entity should be another JSON object with all attributes of the entities. The schema of the attributes should also depend on the type of entity we are dealing with.
- a JSON object representation of the operation executed on the entities, for instance if we want to group persons in the text by family we would send arrays of entities with family relationship (i.e same family name)

We can imagine a graphic interface, where we could click on a found entity to visualize more details about it, and click on a characteristic of the entity and get all other entities with the same characteristic.

## Architecture

Following this quick clarification with the problem let's propose an architecture for our API.



### Controllers

We have defined this module to represent the first interface with our API. It should be able to receive the requests from the user, validate the requests (for example check the size of the text), handle errors and return a response.

We can deal with a variety of requests, such as authentication requests, or any other operation our API should be able to perform.

**JobsController**

This sub-module is in charge of managing different types of jobs the user wants to execute, such as text analysis, image analysis, voice analysis, etc. Hence a job is an operation the user wants to execute on the API.

For now, regarding the needs of this project, we will consider only text analysis jobs, which lead to the following tasks:

- Receive text analysis requests and validate them. Request should be a POST method sending a JSON object. The JSON object should contain:
    - The text: not exceeding a certain size (check size limit of TextRazorAPI).
    - Configuration options: the text analysis methods or algorithms to use in the text analysis task.
    - Output options: the text analysis results expected as a response

- Create and run a text analysis job.
  If we want to be able to process many requests simultaneously (many requests from one or many users) we should perform parallel computing with threads. Hence each request should be a new thread, but the *WSGI* server used by default in *Bottle* is a non-threading HTTP server.
  The easiest way to overcome this is to install a multi-threaded server library like *paste* or *cherrypy* and tell Bottle to use that instead of the single-threaded server. But for now we will use the default *WSGI* server.

- Return a response to the user.

## JobModels

As the user should be able to execute different types of jobs (text analysis, image analysis, etc), this module aims to represent the execution of one kind of a job.

A job is executed according to a workflow. Each step of the workflow represents a state of the job. Hence at each step/state the job will be enriched following the process of the previous step. At the end, the job returns the result.

Considering a request that needs to execute many jobs, we should perform parallel computing with threads. We implement this behavior with *concurrent.futures.ThreadPoolExecutor object.* This object allows us to execute different jobs in parallel, wait for them to end and retrieve each one of their results.

We will focus for now in only one type of jobs, the text analysis job.

**TextJob**

The text job sub-module is in charge of all the steps of the text workflow, following the description of the goals of this API, we can define the following tasks :

- Extract entities from the text: we use the configuration options in the user request to select and execute the right procedure for entity extraction from the given text.

- Enrich entities with information:  collected from different sources (databases, APIs, etc). Once again we use the configuration options to select all the sources desired by the user.

- Aggregate entities according to a strategy: according to the output options defined by the user, we will return the results expected (persons with same job, same age, etc).

- Return the result as a JSON object.

- A transversal task of the TextJob sub-module is to keep track of the information relative to the execution of the workflow. Such as the status (of the job), the starting time, the logs etc. Imagine we want to pause the job and restart it later, or if we want to monitor our application.

## TextWorkflow

This module defines all elements and components that can be used for our text analysis, such as data models, NLP algorithms, tools to target external APIs, etc.

Regarding one task or step of the workflow (entity extraction for example), we may have many elements able to perform the task (TextRazor API or internal NLP methods, etc). Therefore we implement the **strategy design pattern**, to be able to select one or another method.

Also as we will create entity objects while sometimes not knowing what type of entity we are dealing with (a person, an organization, etc), until collecting more information for example, we should implement the **factory design pattern**.

**Entities**

In a workflow we will have to manipulate data models. Here we represent the Entity data model. According to the main goal of our API, we will manipulate Entities. Therefore we need to define a class representing these entities, from which we can inherit an organisation class, a person class, etc. But for now we will only manage entities and persons.

We also need to formalize information about entities through attributes and methods.

#### EntitiesExtractors

This sub-module aims to extract entities from a given text. We can use different methods and each one of them should inherit from this sub-module. The other tasks differ depending on the chosen strategy, and for now we will only implement the tools allowing us to use the TextRazor API.

#### EntitiesResearchers

This sub-module aims to collect information concerning the entities extracted from the given text. Same, it can use different sources and methods to collect information and each one of them should inherit from this sub-module. The other tasks differ depending on the chosen strategy, for now we will only implement the tools allowing us to use the Wikidata API.

#### EntitiesAggregators

This sub-module aims to aggregate entities and study the relations between them. We can imagine different applications but, according to the subject we will focus now in finding common characteristics between the entities extracted from the given text.

### Configs

This module contains configuration files or our API.

#### ServerConfig

In this sub-module we store the server configuration. For instance the default value for the host and port. But moreover we set up the routes a user can target within our API.

#### TextWorkflowConfig

As seen previously a workflow can be configured and here we store the needed configurations at the start of the API, such as default values, keys to connect to APIs, etc.

#### LoggersConfig

A YAML file describing the main logger of the app, the format and its handlers.

### DataManagers

This module manages how data is manipulated and stored  through the API. We can have multiple types of database depending on what we are storing. They could be MongoDB, ElasticSearch, etc. For now we didn’t have time to implement this module.

## Tools & Libraries

For the development of the backend of the API, we have chosen to use Python3.8 in respect with the PEP 8 convention. We installed the following libraries as well:

- **Bottle** : Micro web-framework for running a server.
- **Requests** : Library easing the way we make HTTP requests.
- **Jsonschema** : Library allowing us to validate JSON object format.
- **Pyyaml** : Library allowing us to manipulate yaml files in python.
- **Wikidata** : Library easing the use of the Wikidata API.
- **Coloredlogs** : Library to color the logs on the terminal.

# Conclusion

This project was very interesting and challenging. While I was able to complete the main goals of the API in a short time, there are many directions for improvement in order to put it into production.

First there is a need to implement the DataManager module. We should be able to store data which can be: users and accounts, user activities, jobs, data models (original text, entities, relations, etc), logs, etc.

Second, is to implement more unit tests as long as integration, non-regression tests and test coverage. It would have been better to use the Test Driven Development approach, but due to time constraints I didn't do it.

Then we should work on the stability of the API, by managing resources availability (memory and CPU usage), setting automatic restart (with crontab as a start), and handling sudden crashes and failures (with automatic backups), as well as a better management of errors and exceptions.

We should address the security issues as well. Starting by implementing an authentication method, using for example API keys, credentials, etc.

Another direction is scalability. We have to implement parallelization of the requests, make configuration through environment variables and develop configuration files for deployment to be able to remain on a cloud infrastructure which will manage load balancing.

We can also build a pipeline CI/CD with separated stages (build, test and deploy) to automate deployment.

Finally all these steps will help to put the API in production, maintain and monitor it,  add new features, etc.