Universidade do Minho
Escola de Engenharia
Departamento de Informática

Emanuel Queiroga Amorim Fernandes

Development of a web-based platform for Biomedical Text Mining

December 2019

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Emanuel Queiroga Amorim Fernandes

Development of a web-based platform for Biomedical Text Mining

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Miguel Francisco Almeida Pereira Rocha
Hugo Samuel Oliveira Costa

December 2019

ACKNOWLEDGEMENTS

Agradeço a todas as pessoas que me acompanharam durante o desenvolvimento deste projeto, sem elas não seria possível.
Ao meu orientador, o professor Miguel Rocha, agradeço pela disponibilidade e pelos conhecimentos ministrados que auxiliaram a realizar este trabalho. A oportunidade, apoio e compreensão ao longo do mesmo. Ao meu supervisor na empresa, Hugo Costa, por toda a colaboração prestada, disponibilidade e auxílio em cada etapa do desenvolvimento da aplicação. Ao Rúben Rodrigues pela ajuda prestada, nomeadamente na fase inicial do projeto e esclarecimento de dúvidas. A toda a equipa da SilicoLife por serem pessoas afáveis e de boa disposição.
Aos meus amigos por todos os bons momentos de diversão e descontração como também pelo apoio prestado. A toda a minha família, em especial ao meu pai, José Fernandes, à minha Mãe, Paula Fernandes, e ao meu irmão Paulo Fernandes por todo carinho, apoio, dedicação e motivação. À minha namorada, Catarina Vieira, por me acompanhar e apoiar em todos os momentos da minha vida.

STATEMENT OF INTEGRITY

RESUMO

A mineração de Literatura Biomédica (BioLM) pretende extrair informação de alta qualidade da área biomédica, através da criação de ferramentas/metodologias que consigam automatizar tarefas com elevado dispêndio de tempo. As tarefas subjacentes vão desde recuperação de informação, descoberta e recuperação de documentos relevantes para a extração de informação pertinente e a capacidade de extrair conhecimento de texto. Nos últimos anos a SilicoLife tem vindo a desenvolver uma ferramenta, o *@Note2*, uma BioLM *Workbench* multiplataforma baseada em *JAVA*, que executa as principais tarefas inerentes a BioLM. Também possui uma versão autónoma com uma interface amigável para o utilizador.

Esta tese desenvolveu uma plataforma de software baseada na web, que é capaz de executar algumas das tarefas de BioLM, com suporte num núcleo de bibliotecas do projeto *@Note*. Para tal foi necessário melhorar o servidor *RESTful* atual, criando novos métodos e *APIs*, como também desenvolver a aplicação baseada na web, com uma interface amigável para o utilizador, que comunicará com o servidor através de chamadas à sua *API*.

Este trabalho focou o seu desenvolvimento em tarefas relacionadas com recuperação de informação, focando na pesquisa eficiente de documentos de interesse através de uma interface integrada. Nesta fase, o objetivo foi também ter um conjunto de interfaces capazes de visualizar e explorar as principais entidades envolvidas em BioLM: pesquisas, documentos, corpora, entidades relacionadas com processos de anotações e recursos.

Palavras-chave: Aplicação Baseada na Web, Mineração de Literatura Biomédica, Recuperação de Informação.

ABSTRACT

Biomedical Text Mining (BTM) seeks to derive high-quality information from literature in the biomedical domain, by creating tools/methodologies that can automate time-consuming tasks when searching for new information. This encompasses both Information Retrieval, the discovery and recovery of relevant documents, and Information Extraction, the capability to extract knowledge from text. In the last years, SilicoLife, with the collaboration of the University of Minho, has been developing *@Note2*, an open-source *Java*-based multi-platform BTM workbench, including libraries to perform the main BTM tasks, also providing user-friendly interfaces through a stand-alone application.

This work addressed the development of a web-based software platform that is able to address some of the main tasks within BTM, supported by the existing core libraries from the *@Note* project. This included the improvement of the available *RESTful* server, providing some new methods and APIs, and improving others, while also developing a web-based application through calls to the API provided by the server and providing a functional user-friendly web-based interface.

This work focused on the development of tasks related with Information Retrieval, addressing the efficient search of relevant documents through an integrated interface. Also, at this stage the aim was to have interfaces to visualize and explore the main entities involved in BTM: queries, documents, corpora, annotation processes entities and resources.

Keywords: Biomedical Text Mining, Information Retrieval, Web Based Application.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

AJAX	Asynchronous JavaScript And XML.
API	Aplication Programing Interface.

B

BASE	Bielefeld Academic Search Engine.
BTM	Biomedical Text Mining.

C

CORS	Cross-origin resource sharing.
CSS	Cascading Style Sheets.

D

DAO	Data Access Object.
DOI	Digital Object Identifiers.
DOM	Document Object Model.

G

GO	Gene Ontology.

H

HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.

I

IE	Information Extraction.
IOC	Inversion of Control.

IR Information Retrieval.

J

JDBC Java Database Connectivity.

JSON JavaScript Object Notation.

M

MESH Medical Subject Headings.

ML Machine Learning.

MVC Model View Controller.

N

NCBI National Center for Biotechnology Information.

NER Named Entity Recognition.

NLP Natural Language Processing.

O

ORM Object Relational Mapping.

P

PDF Portable Document Format.

PID Persistent Identifiers.

PMC Pubmed Central.

POJOS Plain Old Java Objects.

POS Part of Speech.

R

RDBMS Relational Database Management System.

RE Relationship Extraction.

REST Representational State Transfer.

S

SQL	Structured Query Language.

U

UM	University of Minho.

W

W3C	World Wide Web Consortium.

X

XML	Extensible Markup Language.

1 INTRODUCTION

1.1 CONTEXT

The Life sciences produce a large amount of information that is spread in scientific publications and databases (Larsen and von Ins, 2010). The publications contain textual non-structured data that turns the search and extraction of high-quality information into a difficult challenge. As a consequence, biomedical researchers spend a large amount of time extracting useful information from literature. *Biomedical Text Mining (BTM)* is the field that applies a process of deriving high-quality information from texts and literature of the biomedical domain. Thus, BTM appeared to create tools/methodologies that can automate and reduce time-consuming tasks when searching for new information in biomedical literature (Shatkay and Craven, 2012).

Information Retrieval (IR) and *Information Extraction (IE)* are the two major BTM tasks. IR includes the methods related to the discovery of documents (articles, books, patents) and their pre-processing to make them suitable for subsequent operations (e.g. *Portable Document Format (PDF)* to text conversion). IE on the other hand, has the capability to extract high-quality information from text streams. Two of the main tasks in IE are *Named Entity Recognition (NER)* and *Relationship Extraction (RE)*.

These two processes can be tested in terms of performance and accuracy, being typically validated against golden standard corpora (set of biomedical documents) that contain annotations (bio-entities and relations) curated by specialists (Kim et al., 2003). In the last decade, challenges such as Biocreative (http://www.biocreative.org/) were created by the BTM community to compare and improve BTM algorithms.

Over the last few years, the Biosystems research group *University of Minho (UM)* and the SilicoLife company have worked in the BTM field. In this period, *@Note2* [1] a multi-platform BTM Workbench has been developed, fully written in *Java* and that uses a *MySQL* database. This framework copes with the most important IR and IE tasks in the field and promotes multi-disciplinary research. The *@Note2* architecture, based in plug-ins, enables

1 http://www.anote-project.org

the fast development of new BTM methodologies, while keeping a user-friendly interface (Lourenço et al., 2009).

The software developed in the *@Note* project includes a set of core *Java* libraries implementing the main tasks in IR and IE. In the last couple of years, these have been used as the basis for the implementation of a server, providing a number of BTM methods, with two different options, through well-defined *RESTful Aplication Programing Interface (API)*s or integrated database access. A set of graphical user interfaces are provided within the *@Note* client stand-alone application, which is made available as open-source software.

1.2 OBJECTIVES

The main goal of this work is the development of a web-based software platform that is able to address some of the main tasks within BTM, focusing on the visualization and exploration of the main entities and results (e.g. *documents*, *corpora*, *annotations*). This will be supported by existing core BTM libraries from the *@Note* project.

In detail, the scientific/ technological objectives are:

- To review methods/algorithms and existing tools within the field of BTM;
- To review the existing software within the *@Note* BTM framework;
- To review web programming tools, selecting the most adequate for this project;
- To improve the available *RESTful* server within the *@Note* project, providing new methods and APIs, and improving others making them more suitable to support web-based applications;
- To develop a web-based application allowing the user to visualize, explore and manage the main entities involved in BTM.
- To develop a tool allowing the Boolean search of *documents* by named entities and by free text with an interactive and intuitive web-based interface integrated, and allow *corpus* creation by both search types.

1.3 STRUCTURE OF THE THESIS

The state of the art (next chapter) explores the BTM field, focusing on topics such as the problems it addresses, its main tasks and the resources that support them. It reviews some developed tools by the scientific community that implement BTM methodologies similar to the proposed objectives. Also, it briefly introduces the *@Note* project where this thesis is integrated in. The chapter describing the web application development follows, explaining the architecture and domain of the solution, and describing the development of the web application starting by the back-end implementation. The next chapter focuses on the results, showing the web application interface with some examples of data and interactions. The last chapter brings the main conclusions of the work and some lines of prospective future work.

2 STATE OF THE ART

2.1 BIOMEDICAL TEXT MINING

The significant increase in the number of unstructured biomedical texts in the last few years gave birth to BTM. Resorting to computational tools, time-consuming tasks can be replaced with automated processes, designed for quality knowledge extraction in the biomedical and molecular biology domains (Shatkay and Craven, 2012; Cohen and Hunter, 2008).

Tasks related to BTM can address a vast number of purposes giving answers to differentiated groups within the scientific community (Shatkay and Craven, 2012). The tasks can be divided in two distinct categories: IR, where the main purpose is to find and organize relevant documents from a larger set, through well-formed queries on an indexed database, and IE which is responsible for more in-depth analyses of text, such as NER where the purpose is to identify interesting entities like genes, proteins or cellular locations, while RE identifies links between these entities.

The NER process can be performed by distinct approaches, including lexicon-based, rule-based or *Machine Learning (ML)* based techniques. Each approach has advantages and disadvantages regarding the need to resort to manual curation (e.g. necessity of curated lexica, rules or ML models).

The RE process also has different approaches (e.g. semantic rules, *Part of Speech (POS)* trees), but ML methods are still an option. RE methods identify possible relations between previously annotated bio-entities (Shatkay and Craven, 2012; Rodriguez-Esteban, 2009).

Evaluation of the processes pipelines associated with a specific task is done with a standard goal. The standard goal is a corpus with instances of the task, ideally providing correct solutions free of errors, although this scenario rarely happens. The evaluation is based on metrics that score the processes, and based on the results responds to questions about how well the system performs (Shatkay and Craven, 2012).

Typical performance measures for both IR and IE systems are recall (Rc) and precision (Pc):

$$recall = \frac{TP}{TP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

Recall and precision are calculated by dividing the original set the BTM task processed, into four subsets related to its classification. True positives (*TP*) are the entities correctly classified, false positive (*FP*) are incorrectly classified entities, true negatives (*TN*) are entities correctly classified as negative or correctly not classified as an entity, and false negatives (*FN*) are when the system failed to classify an entity or incorrectly classified as negative. Recall generally corresponds to *sensitivity*, however reaching 100% of this metric is trivial. For example, if an IR system labels all retrieved documents as relevant it would successfully indicate the user all the documents relevant to him, however a system like this would be useless. A system with a nearly perfect precision is also trivial to implement, modifying the previous IR example to a system that only labels one document as relevant. To more systematically evaluate a system, two additional metrics that balance the trade-off between precision and recall are used. They are the accuracy and *F-score*:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Fscore = \frac{2Pc * Rc}{Pc + Rc}$$

2.1.1 *Information retrieval*

In any research, the first step is to obtain relevant information, from where knowledge can be extracted. IR offers several methods to search for documents that might be interesting to the users needs, the goal is to retrieve all relevant documents, while correctly discarding irrelevant ones. Sometimes, the number of relevant documents might require time to analyze that greatly surpasses what is available. When this happens, a finer grained query can be applied or the information can have some priority enabling structure, showing the user the retrieved documents which best suit him.

Publications can be accessed on public repositories, aggregators or catalogs, that can contain abstracts, citations, full text articles, reports, etc. To prevent ambiguity, and enable easy reproduction of results reported by other teams, each resource has a persistent identifier. Two examples are: Handle *Persistent Identifiers (PID)*, where abstract ids are assigned to a resource in accordance to the Handle schema, and *Digital Object Identifiers (DOI)*[1], serial ids used to uniquely identify digital resources (Przybyła et al., 2016).

1 http://www.doi.org/

Currently, two of the most popular repositories are *Pubmed Central (PMC)*[2] and arXiv[3]. PMC contains around 3.9 million publications related to Biomedical and life sciences fields. It also contains an open access subset, that is part of the total collection of articles in PMC, with over 1 million articles. ArXiv gives access to full preprints and abstracts of 1.2 million publications related to Biology, Chemistry, Physics, Computer science and Mathematics, among others.

Aggregators are especially useful to text miners, as they contain information harvested from multiple sources searchable and available in an uniform way. The *Bielefeld Academic Search Engine (BASE)*[4] by Bielefeld University Library provides more than 100 million documents from more than 4,000 sources and about 60% of the indexed documents for free (Open Access). It contains abstracts, full text articles, books and multimedia documents, software and datasets.

Focused on the area of the Life Sciences, we have the aggregator PubMed `https://www.ncbi.nlm.nih.gov/pubmed/` by the *National Center for Biotechnology Information (NCBI)*, U.S. National Library of Medicine, that comprises more than 30 million citations for biomedical literature from MEDLINE, life science journals, and online books. Generally, full text articles are not available, but may contain links to their content or publisher web sites. However, NCBI's PMC archive contains more than 5 million free full-text records, of biomedical and life science research `https://www.ncbi.nlm.nih.gov/pmc/`.

An indexed structure is fundamental to retrieve information from these databases, especially for Boolean queries. The structure relies on tokenization, a *Natural Language Processing (NLP)* process of segmenting text, where the document(s) are divided into tokens later combined to form terms. Terms are sorted, typically alphabetically, and they reference documents containing them forming an index. When a Boolean query is executed, the documents referenced by the terms which satisfy the set operation from the query are retrieved. However, these terms alone may not be sufficient as they only contain tokens from the documents processed. For example, querying a term not present in the index but where a synonym is, the documents referenced by the synonym should be retrieved which will not happen. To solve this problems, other terms corresponding to fundamental concepts may be inserted.

This type of index can be enriched with different types of useful information to support processes of categorization or summarization, among others:

- The exact position on the text or words surrounding the term, which is useful for longer phrases identification and location.
- Grammatical information, such as part-of-speech tags.

2 https://www.ncbi.nlm.nih.gov/pmc/
3 https://arxiv.org/
4 https://www.base-search.net

- Statistics, like the number of times and number of documents a term occurs and is contained, respectively.

Simple Boolean queries, despite efficient implementation have domain-specific limitations problems, such as the fact that short Boolean queries might result on impractical number of documents retrieved. Natural language problems can lead to the retrieval of irrelevant documents, while relevant ones may be missed.

Methods based on the vector model try to address the issues of Boolean queries explicit words and terms dependency. Documents are viewed as vectors of weights, reflecting the significance of the individual terms. Queries also follow this principle and are also represented as a vector. The search algorithm can be reduced to comparing similarities between the documents and query vectors, being the most similar retrieved (Shatkay and Craven, 2012).

2.1.2 *Named entity recognition*

NER contains the methods of IE responsible for the identification of interesting entities in texts. Before explaining the processes of NER, let's enumerate some factors related to this task, which impose significant difficulties:

- Synonyms — the same entity (e.g. gene) can have different names depending on authors and context;
- Abbreviations — the same entity (e.g. compound) can have several different abbreviations and acronyms depending on the context.
- Homonymous — biological entities can be represented by names that have common meanings in English (e.g. a gene name FOR).
- Ambiguity — the same name is often used for entities of different classes (e.g. a gene and its encoded proteins).
- Entity names composed by combining others (e.g. genes or compounds).

Dictionary based methods

On this method, a dictionary structure is parsed searching for matches with entity names kept in the dictionary keys and possible synonyms. This process has two major limitations: the results will only be as complete as the dictionary used, i.e. an entity that is not present on the structure cannot be matched, and context is not taken into consideration, so the homonymy problem is not addressed, i.e. common terms used in natural language should be omitted from the dictionaries as they will generate a huge number of false positives.

One approach to mitigate the problems of an incomplete dictionary is to use a generalized dictionary relying on named patterns. This type of dictionaries might increase recall; however, some precision is typically lost. Different kinds of dictionaries can be employed as components of the NER, with better evaluation than the separate parts.

Rule-based methods

These methods rely on morphological and lexical properties of named entities of different classes, as well as syntactic analyses. The core rules can be devised separated in stages. One method by Fukuda et al. (1998) has 2 stages of rules, in the first stage the terms are identified, and the second stage extends them with rules based on syntax like POS. These methods have the advantage of how easy the extraction process is to understand, while their drawback is the non existence of default rules. According to the situation, rules might have to be modified and created, and therefore this task typically expends significant human effort.

Learning classification models

Machine Learning methods are automatic processes where the model learns from labeled training data. These models automatically identify regularities among named-entities if a good dataset of training examples is available. Generally, producing these datasets is very costly and requires a considerable amount of manual work. Maximum-entropy models, probabilistic sequence models, like Hidden Markov Models and Conditional Random Fields are some examples of machine learning methods widely used for NER tasks.

2.1.3 *Relation extraction*

This field of BTM also includes methods for the identification of relations between entities of interest. The extraction of relations is a complex task and the systems are mostly based on hand-coded patterns, ML processes or a combination of the two, using detailed syntactic analysis and semantic categories and relationships.

Co-occurrence

This kind of methods are based on finding co-occurrences of two entities within short passages of text or in documents. As a standalone process, it has a major limitation, just by two entities co-occurring, a relation is not confirmed as the relation can be elsewhere on the text or even be non existent. To support the hypothetical existing relationship, the frequency with the possibility by chance in mind and textual context can provide more information towards the significance of the relationship found.

Rule-based methods

On these systems, rules can be applied on syntactic and semantic level to pre-processed text divided into tokens. Interesting entities like proteins and verbs can be labeled using NER and POS tagger processes, respectively.

POS taggers can be based on models that are induced by supervised machine-learning methods from labeled corpora. These ML methods are able to identify statistical regularities in the context of a given token that enable its part of speech to be disambiguated (Shatkay and Craven, 2012).

2.1.4 *Tools*

The scientific community has at its disposal several resources available for IR, IE processes, as well as access to multiple structured databases to use on their work. Some function as a standalone service, others provide a workflow of operations combining existing resources. Next, we enumerate some tools that implement BTM methodologies:

- XplorMed[5] (Perez-Iratxeta et al., 2001) — The XplorMed server allows to explore a set of abstracts derived from a MEDLINE search. The system gives main associations between the words in groups of abstracts. Then, you can select a subset of your abstracts based on selected groups of related words and iterate your analysis on them.
- Whatizit[6] — Text processing system that allows text mining tasks on text, great at identifying molecular biology terms and linking them to publicly available databases. Several vocabularies can be integrated in a single pipeline, examples of already integrated vocabularies are Swissprot, the *Gene Ontology (GO)*, the NCBI's taxonomy, Medline Plus.
- Egas[7] (Campos et al., 2013) — Egas is a web-based platform for biomedical text mining and collaborative curation, supporting manual and automatic annotation of concepts and relations.
- PubAnnotation[8] — PubAnnotation is a repository of text annotations, especially those made to literature of life sciences, e.g., PubMed or PMC articles. If one has such annotations, they can be registered in PubAnnotation. When annotations are registered, PubAnnotation aligns them to the canonical text that is taken from PubMed and PMC, which means all the annotations in PubAnnotation are linked to each other through

5 http://xplormed.ogic.ca/
6 http://www.ebi.ac.uk/webservices/whatizit
7 https://demo.bmd-software.com/egas/
8 http://www.pubannotation.org/

canonical texts. It is a new way of publishing or sharing text annotations using recent web technology: annotations will become accessible and searchable through standard web protocol e.g., *Representational State Transfer (REST)* API.

- Argo[9] (Rak et al., 2012) — Argo is an inter-operable, integrative, interactive and collaborative system for text analysis with a convenient graphic user interface to ease the development of processing workflows and boost productivity in labour-intensive manual curation. Robust, scalable text analytics follow a modular approach, adopting component modules for distinct levels of text analysis. The user interface is available entirely through a web browser that saves the user from going through often complicated and platform-dependent installation procedures. Argo comes with a predefined set of processing components commonly used in text analysis, while giving the users the ability to deposit their own components.

Table 1 provides some key points on the tools described above.

2.1.5 *Resources*

To support the BTM tasks explored in subsections 2.1.1, 2.1.2, 2.1.3, several resources have been developed over the years. Lexical resources collect frequent lexical information on text corpora such as parts of speech and spelling variants. Terminology resources collect names of entities employed in the biomedical domain, they generally store synonyms and have a hierarchical organization, such as graph structures. Ontology resources store relations, such as *is a* and *part of*, between biomedical entities and their significance Bodenreider (2006). Table 2 presents some of these resources and summarily explains them.

2.2 THE @NOTE PROJECT

The Biosystems research group (UM) and SilicoLife company released *@Note2*[14] currently on version 2.1.4 (November, 23rd - 2016), a BTM tool which promotes the joint efforts between three distinct classes: biologists, text miners and software developers (Lourenço et al., 2009). *@Note* is built on top of *AIBench* a facilitator to the referred multidisciplinary approach empowering the project with modular and flexible development. Its plug-in based approach also enables the fast development of new BTM methodologies, while keeping a

9 http://argo.nactem.ac.uk/
10 https://www.nlm.nih.gov/research/umls/
11 http://wordnet.princeton.edu
12 http://wordnetweb.princeton.edu/perl/webwn
13 http://www.geneontology.org/
14 http://anote-project.org/

Table 1.: Key points on tools that implement BTM methodologies

Tool	Developers	Main focus	Web interface implementation	IR tasks	IE tasks	limitations
xplormed	Carolina Perez-Iratxeta, Peer Bork, and Miguel A. Andrade	Abstracts view can be condensed by relation extraction between words contained on them.		Classification of abstracts based on *Medical Subject Headings (MeSH)* terms;Computation of relations and words to attain a desirable subset.		Server is limited to 500 abstracts; Direct queries to Pubmed are currently unavailable.
Whatizit	European Bioinformatics Institute[a]	Modular server-based solution for literature analysis			Pipeline of several available modules for NER and RE.	On the web interface only one pipeline is available
Egas	Bioinformatics group[b] at The University of Aveiro[c]	Collaborative features, annotation-as-a-service solution.	HTML5, CSS3 and JavaScript	List of identifiers/ query of/to PubMed or PubMed Central	ML-based protein name recognition,protein-protein interactions, etc.	
PubAnnotation	Mainly developed and maintained by DBCLS[d]	Collect annotations to life science literature, free repository of text annotation.	Ruby on Rails	List of identifiers of some major public databases. SPARQL queries.	Has some predefined annotators or let's the user specify one (URL, POST/GET)	Annotations need to be in the PubAnnotation JSON format
Argo	The National Centre for Text Mining[e] at The University of Manchester[f]	Ease on combining components,manual intervention,web-based interface, user collaboration.	Google Web Toolkit and Scalable Vector Graphics	Kleio: a knowledge-enriched IR system for biology (Nobata et al., 2008)	Well structured pipeline workflow with many components.	

a http://www.ebi.ac.uk/
b http://bioinformatics.ua.pt/
c https://www.ua.pt/
d http://dbcls.rois.ac.jp/
e http://nactem.ac.uk/
f http://manchester.ac.uk/

Resource	Type	Summary
UMLS[10] Specialist lexicon (Browne et al., 2003)	Lexical	The SPECIALIST lexicon is a syntactic lexicon of biomedical and general English words, providing orthographic, morphological and syntactic information about individual vocabulary items. The Lexicon consists of a set of lexical entries with one entry for each spelling or set of spelling variants in a particular POS.
Wordnet[11] (Miller, 1995)	Lexical	Wordnet links English nouns,verbs, adjectives, and adverbs to sets of synonyms that are in turn linked through semantic relations that determine word definitions. Its structure makes it useful for NLP processes where synonym words are grouped into unordered sets called synsets. Synsets are linked between each other through relations, like super-subordinate relation (e.g. table *is a* piece of furniture) or cross-POS references (e.g. painter is the AGENT of paint, while picture is its RESULT.). The result network can be navigated via browser[12] or freely retrieved from https://wordnet.princeton.edu/wordnet/download/.
Gene ontology[13] (GO) (GeneOntologyConsortium, 2004)	Ontological	Divided into three distinct structured ontologies that describe gene products in terms of their associated **biological processes** , **cellular components** and **molecular functions** in a species-independent manner. The GO ontology is structured as a directed acyclic graph where each term has defined relationships. Every term has a term name and a unique zero-padded seven digit identifier, a namespace that identifies to which ontology it belongs. All terms except root terms (e.g. biological_process) include links that captures how the term relates to others. Some extra information like synonyms can also be included.

Table 2.: Table with resources available for BTM tasks

friendly user interface, a crucial aspect for creating a bridge between biologists and BTM computational processes.

@Note provides the currently most important tasks of IR and IE in the field implemented on a set of *Java* libraries with well-defined *RESTful* APIs. This collection of methods can be accessed through a *Java* project called *anote2daemon* which can be deployed on a server as a web service using the *JavaScript Object Notation (JSON)* format for communication. However, these methods are not only available through their *RESTful* APIs, each method also has an implemented API within the *Java* project with database access.

2.2.1 *@Note application*

Regarding IR, well defined queries can be produced to search over databases (e.g *PubMed* and *patent* search). From these, documents' meta-data including abstracts are retrieved from the database. If free full text is available, it can be downloaded using an implemented automatic operation. There also options to provide the possibility to add annotations or manually classify the document relevance. To further help management and organization, each document has labels related to context associated like blood testing or Humans. There is also an option to perform clustering over the queries using different methods.

Retrieved documents can be selected to form a corpus, where IE processes can be applied to extract knowledge from the collection. For these processes, the resources currently supported are: dictionaries, lookup tables, rulesets, ontologies, and set of lexical words.

Manual curation is also available for automatic processes, providing an environment where annotated terms in documents are marked with different colors based on their class, helping experts to correct possible mistakes and inconsistencies on their resources.

Table 3 features the NER tasks implemented in *@Note* with a small description of their major role.

For RE processes, *@Note* uses a pre-processed corpus with NER tasks already performed. RE processes are divided into *Corpus Relation Extration* and *Corpus Relation Co-ocorrence Extration*.

Corpus Relation Extration is based on NLP algorithms of POS shallow parsing, using POS-Taggers like *GATE*[15] and *Ling Pipe*[16]. The process can be configured with the following parameters:

- Relation Extraction Model — model for the results like *Binary verb limitation*(1×1) where the verb relation is binary or *verb limitation* ($M \times M$).
- Relation types — Relation types for extraction (e.g compound–compound, compound–biological process).

15 https://gate.ac.uk/
16 http://ir.exp.sis.pitt.edu/ne/lingpipe-2.4.0/

NER task	Major role
ABNER	URL: http://pages.cs.wisc.edu/~bsettles/abner/ Biomedical Named Entity Recogniser. It uses machine learning (linear-chain conditional random fields, CRFs) to find entities such as genes, cell types, and DNA in text.
Chemistry Tagger	URL:https://gate.ac.uk/sale/tao/splitch21.html#sec:parsers:chemistrytagger Designed to tag a number of chemistry items in running text. Currently, it tags compound formulas (e.g. SO_2, H_2O, H_2SO_4 ...) ions (e.g. Fe^{3+}, Cl^-) and element names and symbols (e.g. Sodium and Na).
Linneaus Tagger	URL:http://linnaeus.sourceforge.net/ Performs NER processes based on lexical resources (Dictionaries, Lookup Tables, Rule Sets, Ontologies) matching terms in the text using the Linneaus Tagger. Lexical resources and options can be based on previously applied NER processes.
Lexical resources [native]	Performs NER processes based on lexical resources (Dictionaries, Lookup Tables, Rule Sets, Ontologies) matching terms in the text. Also, results can be filtered using Stopwords (Lexical Words Resource) to remove common English words and improve results using the disambiguation process. Lexical resources and options can be based on previously applied NER processes.

Table 3.: Table with implemented NER tasks on *@Note*

- Advanced model options
  - Using a maximum word distance between verbs (clues) and annotated entities in the sentence.
  - Keep Only Relations where verbs are associated only with nearest entities.
  - Keep Only relations where the entities are only associated to the nearest verb.
- Manual Curation (from other process) — merge previously manually annotated entities from another previous process in the RE process.
- Verbs lists — Use verb list from the lexical resources to filter the results and/or another to add as relation clues.

Corpus Relation Co-ocorrence Extration is based on entity proximity (e.g terms in the same sentence). The model for this process can have the entities in the sentence continuous or mixed in pairs, *Entity Sentence Continuous* or *Mix Entity Pairs Sentence* respectively.
@Note also has a ML based module, the *BioTML* that enables users to create models, trained based on available annotations (NER or RE) created by the user using the manual curation environment.

3 WEB APPLICATION DEVELOPMENT

This chapter starts by describing the technologies and languages used on the development of the *@Note web* application. This chapter then explains the architecture of the system, properly identifying the entities involved and their relations. Since the solution is focused on access and navigation of information, data access and storing is explained in more detail. The major *back-end* implementations are explained in detail, such as, the creation of the *Lucene* database and queries using that database. The final section is dedicated to the *angular front-end* development.

3.1 TECHNOLOGIES AND LANGUAGES

This section explains some technical aspects of the technologies and languages used to develop the application, as well as the methodologies and good practices associated with them to design and implement the system.

3.1.1 *Java*

Java is an object oriented programming language and is used in the back-end implementation of the system. Some of the advantages of programming in *Java* are (Gosling and McGilton, 1995):

- **Portability** : Applications written in *Java* run without modification on multiple operation systems and hardware architectures.
- **Interpreted, threaded and dynamic**: The *Java* interpreter can execute *Java* bytecodes directly on any machine to which the interpreter and run-time system have been ported. *Java* also supports multithreading at the language level and the language and run-time system are dynamic in their linking stages.
- **Object oriented**: *Java* provides a clean and efficient object-based development platform.

- **Robust**: *Java* features help on acquiring reliable programming habits and its extensive compile-time checking, followed by a second level of run-time checking, increases software reliability.

3.1.2 *Hibernate ORM*

An *Object Relational Mapping (ORM)* system works around the object-relational impedance mismatch, the tabular representation data versus the interconnected graph of objects representation by object-oriented languages, by whisking data to and from a relational database to appropriate objects (O'Neil, 2008; hib, a). This mismatch between the two representations has the following problems (hib, a):

- **Granularity**: Object model classes sometimes don't have a one to one correspondence with the tables in the database.
- **Subtypes (inheritance)**: *Relational Database Management System (RDBMS)* do not define anything similar, at least standardized, to inheritance in object-oriented languages.
- **Identity**: RDBMS primary key versus object equals method or == to define the notion of 'sameness'.
- **Associations**: RDBMS foreign keys association representation versus references by object-oriented languages.
- **Data navigation**: In relational databases, to minimize the number of *Structured Query Language (SQL)* queries, several entities are typically loaded via *JOINs* and the target entities are selected before walking the object network.

In addition to helping with this mismatch, *Hibernate ORM* also consistently provides a superior performance over straight *Java* database connectivity code and is highly configurable and extensible (hib, a).

3.1.3 *Hibernate Search and Lucene*

In a digital world, where knowledge is increasingly more vast and differentiated, it is urgent we empower ourselves with tools that allow us a fast access to this knowledge. *Lucene* is a high performance text search library written in *Java*, already used by *@Note* to search resources and its elements. At *@Note*, *Lucene* is used through *Hibernate Search*, that integrates *Apache Lucene*, handles indexing, datastore synchronization, clustering and infrastructure transparently, also providing an API for query building.

Hibernate search offers full text search for objects stored by *Hibernate ORM* and other sources. It has the following advantages (hib, b):

- Offers control on how to store data, and how to extract information, also exposing all capabilities of *Apache Lucene*.
- Offers architectural solutions to maintain a high performance, scalable and distributed index.
- Query results can be organized by groups and categories.

Model

At the beginning of a search, it is necessary to have a database with efficient methods to arrive at the intended information, allowing for fine-grained queries with options like case-sensitivity, suggestions, filtering of results and sort. *Lucene*'s index stores documents' terms into maps, called inverted indexes because the search is based on the terms of the documents. Each term must have a token and a field name.

The way query results are organized can be explained using the *Lucene Practical Scoring Function*[1] :

$$score(q,d) \;=\; \sum_{t\ in\ q} \left(tf(t\ in\ d) \;\times\; idf(t)^2 \;\times\; t.getBoost() \;\times\; norm(t,d)\right)$$

1. **tf(t in d)** is the term's frequency, so documents that have more occurrences of a given term receive a higher score.
2. **idf(t)** correlates to the inverse of *docFreq* (the number of *documents* in which the term t appears), rarer terms give higher contribution.
3. **t.getBoost()** is a search time boost of term t in the query q.
4. **norm(t,d)** is an index-time boost factor that solely depends on the number of *tokens* of this field in the *document*, so that shorter fields contribute more to the score.

Hibernate Search model

To map the domain model, persistent classes are marked with several annotations and filled with value types that define their state. To add free text capabilities with *Hibernate Search*, additional annotations to the class are required. Table 4 lists some of the referred annotations.

Analysis is the process in which text is converted into terms, *Hibernate Search* has an configurable *analyser* class that can be applied to entities, properties or even fields. Custom reusable *analysers* can be defined and are comprised by: a name, unique identifiers of the

1 https://lucene.apache.org/core/8_2_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

Annotation	Location	Use
@Indexed	Class entity	Specifies that the entity is indexable, can have searchable fields.
@AnalyzerDef	Class entity	Definition of a custom *analyzer* for indexation.
@AnalyzerDefs	Class entity	Allows multiple *@AnalyzerDef* definitions.
@Field	Basic value type	Specifies that the field is searchable. A custom or built-in-in *analyzer*, can be passed to the field to customize the indexation.
@Fields	Basic value type	Allows multiple *@Field* annotations to be associated to the value.
@SortableField	Value type	Explicitly specifies that the field is sortable
@IndexedEmbedded	Collection value type	Index associated entities for example: (*@ManyToMany*, *@*ToOne*, *@Embedded* and *@ElementCollection*) as part of the owning entity
@FieldBridge	Value Type	Custom *bridge* for the field for more fine-grained indexing.

Table 4.: Some *Hibernate Search* annotations applicable on data persistence classes

analyser definition; a list of character filters, responsible for the pre-processing of characters before they are converted into tokens; a *tokenizer*, transforms the input characters into tokens; a list of filters that can remove, modify or add words. This separation of tasks enables re-usability of these components when defining other *analysers*.

Query

Hibernate Search provides a list of querying methods that can be combined to make a fine grained search. Some of the options available are:

- Keywords;
- Words where some are unknown;
- Phrases;
- Range values, like between dates, numbers or words;
- Results similar to the text search.

Queries can also be configured with the following options:

- *boostedTo* (on query type and on field): boost the whole query or the specific field by a given factor.
- *withConstantScore (on query)*: all results matching the query have a constant score equal to the boost.

- *filteredBy(Filter) (on query)*: filter query results using the Filter instance.
- *ignoreAnalyzer (on field)*: ignore the analyzer when processing this field.
- *ignoreFieldBridge (on field)*: ignore field bridge when processing this field.

3.1.4 *Spring framework*

The *Spring* framework is a *Java* platform that provides comprehensive infrastructure support for developing *Java* applications. *Spring* enables to build applications from *Plain Old Java Objects (POJOs)* and to apply enterprise services non-invasively to POJOs. Two of the advantages of the *Spring* platform are: turn a local *Java* method into a remote procedure without having to deal with remote APIs; make a *Java* method execute in a database transaction without having to deal with transaction APIs. (Pivotal, 2017)

The *Spring* framework consists on features organized into modules, grouped into containers such as (Pivotal, 2017):

- ***Core Container***: Has modules that provide the fundamental parts of the framework, such as the *Inversion of Control (IoC)* and dependency injection features. It also provides access to objects in a framework-style manner.
- ***Data Access/Integration***: Has transaction modules for integration, with *Hibernate* for example, and modules for data access, for example giving support to pragmatic and declarative transaction for classes. It also provides a *Java Database Connectivity (JDBC)* abstraction layer.
- ***Web***: Contains *Spring*s *Model View Controller (MVC)* and REST Web *Services* implementation.

Inversion of Control

In a *Java* platform application development, it is the developer's task to organize basic building blocks into a coherent whole. This task can use patterns, good practices given a name, a description of what it does, where to apply it, and the problems it addresses. The *Spring Framework* codifies formalized design patterns as first-class objects that can be integrated in the application. IoC also known as dependency injection, provides formalized means of composing disparate components into a fully working application.

IoC is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments. These dependencies are injected when creating a *bean*. In *Spring*, the objects that form the backbone of the application, and that are managed by the *Spring* IoC container, are called beans (Pivotal, 2017).

3.1.5 *Web-based technologies*

With the advances on modern browsers, we observed an increase on web-applications, where deployed *JavaScript* code size has grown 45% over the course of the year 2015 (ENACHE, 2015). Web applications make the project more accessible, avoiding possible complicated installation and configurations to use the product. However, writing a full-fledged application on *JavaScript*, *Cascading Style Sheets (CSS)* and a general-purpose scripting like *PHP* alone lacks productivity (e.g. duplication of code, writing too much code), maintainability (e.g. lack of modular components) and abstraction.

To help solve these problems, several web frameworks are available, either developed with an MVC architecture in mind like *Ruby on rails*[2], or their well designed framework are MVC enablers like *Django*[3].

MVC is a design pattern that encourages the division of the application into three interdependent components. *Views* contain how the data are presented, among the components needed for display they might also contain *subviews*. *Controllers* are the interface between their associated *views* and *models*, they define how the application interacts with the user. The *model* is the application central structure, contains all of the application logic and data. A modular structure is extremely useful, isolating functional units as much as possible, facilitating their understanding and modification (Krasner and Pope, 1988).

Angular

Angular[4] is a web framework, 100% *Javascript* and client-side. Powered by google, created by Miško Hevery. Some examples built with it can be found at `https://www.madewithangular.com/` with websites like the one from the Oscars[5].

Angular is extremely modular, actually many of its libraries are *modules*, as well as, many other third party libraries are also available as *modules*. An *angular* application must have at least one *module*, *the root module*. From this *module* the application can be constructed by adding other cohesive *modules* of functionality, they can feature a specific business domain (e.g. a *module* for IR tasks) or simply provide some utility (e.g. a *bootstrap module* to help with the application graphical design).

Two special features about *angular* are: *Data binding* and *Directives*. *Data binding* is the automatic synchronization of data between the *model* and *view components*. The *view* can be thought as an instant projection of the *model*, which enables the *controller* to be completely separated and unaware of the *view*'s existence. This is especially useful for testing and applying different presentation forms of information to the same *controller*. *Directives*

2 http://rubyonrails.org/
3 https://www.djangoproject.com/
4 https://angularjs.org/
5 http://oscar.go.com/

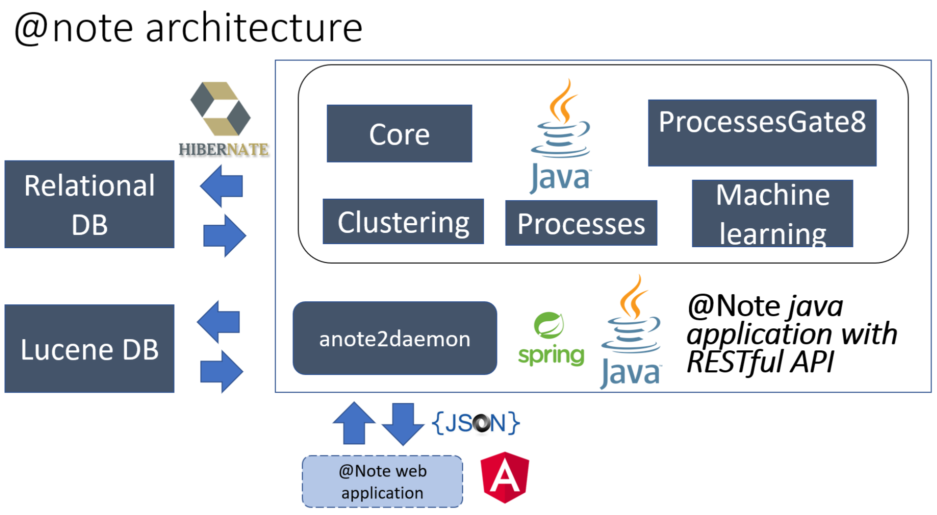


Figure 1.: *@Note* Architecture

are markers on a *Document Object Model (DOM)* element that extend the *HyperText Markup Language (HTML)*, the language for describing the structure of web pages (w3h), with new attributes, providing functionality to the application. The DOM is an API for valid HTML and well-formed *Extensible Markup Language (XML) documents*. It defines the logical structure of *documents* and the way a *document* is accessed and manipulated (Wood and Level, 2004). *Angular* already provides many useful built-in *directives* (e.g. *ng-model*, *ng-repeat*) and encourages the user to write their own when needed. For example, with the *ng-repeat* directive, we can mark a table HTML tag to populate a table bind to a *model*. We write the code to populate the columns, and one row and the *directive* will repeat the rows as many times as there are elements in the *model*.

3.2 @NOTE ARCHITECTURE

In figure 1, we provide the architecture of the *@Note* system, where we can see that the developed web application data is a *RESTful Java* application connected to selected databases. The project has two databases to query for information, both implemented using *Hibernate*, and set up through configuration files. One of them is a relational database, where we used *MySql*, and the other is a repository of files for *Lucene* queries. The *anote2daemon* is a *Spring RESTful* web service, giving access to *@Note's* methods to the outside. The mapped methods use the five core *@Note libs* to access the database and execute BTM processes. Finally, the *@Note* web application has services implemented with REST requests configured to query a server with an *@Note* web service running.

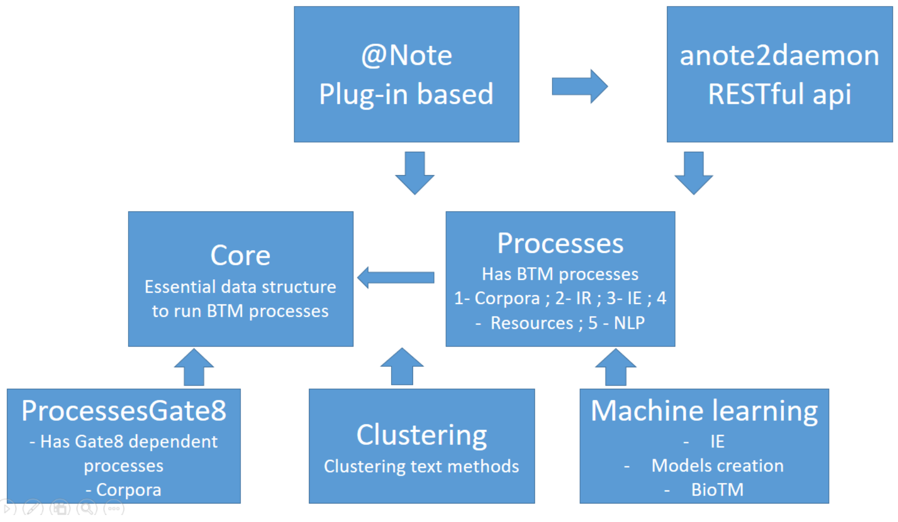


Figure 2.: *@Note* java libraries

3.2.1 *@NoteLibs*

The *@Note* project has seven *Java* libraries connected and maintained as *Maven*[6] projects. Figure 2 shows what each library contains and their dependencies.

The more relevant *Java* libraries to the web application development are: *Core*, *anote2daemon* and *Processes*. The *Core* library has the essential data structures, data access and business logic of the domain. The *Processes* library implements the majority of the BTM processes. Processes implemented in other libraries have a reason to be in a separated library, for example *ProcessesGate8* has the BTM processes dependent on the *Gate8* framework. The *anote2daemon* library provides a *RESTful* API to access *@Note's* data and BTM processes.

The contributions of this thesis to the improvement of the *@NoteLibs* can be summarized as:

- *Core*:
  - Use *Hibernate Search* to create a *Lucene* database and queries for that database. The *Resource* and *ResourceElements* already had implementations related to *Lucene*.
  - Implementation of attribute languages for Boolean expression search.
  - Implementation of data access methods better suited for the web application, for example, implementation of paginated results.
  - Implementation of data access to track the progress of running processes and consult reports of finished processes. Data access implementation involved creating a data persistence class, services and the *Data Access Object (DAO)*.

6 https://maven.apache.org/

- ***Processes***: Added two *corpus* creation methods, using the process tracking implementation at the *Core*. One method adds publications to the *corpus* from a set of publication identifiers, and the other from a set of objects from the classes implementing the interface *IPublication*.

- ***anote2daemon***:
  - Added methods to the controllers related to the new implementations on *Core* and *Processes*.
  - Added *beans* of the new implemented services at Core.
  - Implemented caching for some methods.
  - Implemented methods related to authentication and communication with the *@Note* web application.

3.2.2 *@Note data access*

The *@Note Core* project provides access to the entities stored in the relational database and the *Lucene* database. Figure 3 has a relationship between the entities and the modules they are included in. Access is granted by a class implementing the interface *IDataAccess*, and all methods that access the database must be defined here. Two methods implement *IDataAccess*:

- *DatabaseAccess*: Reads configurations of the relational and *Lucene* databases, accessing the files and creating an *Hibernate* session. Implements the methods to access the database using the services in *dataaccess.implementation* and *Lucene* packages.

- *DaemonAccess*: Has methods to connect and authenticate for an *anote2daemon* running at a server. In the *anote2daemon*, all methods declared at *IDataAccess* are implemented, by *Hypertext Transfer Protocol (HTTP)* requests.

Services are the bridge between the DAO and data access, they request the information from methods implemented in DAOs and return business objects.

Lucene DAO implementations use the *Hibernate Search* API to build database queries to access the *Lucene* index of *@Fields* defined in data persistence classes at *model.entity*. Model DAO implementations use *Hibernate* to query the relational database.

GenericDao for both *Lucene* and *model* can be instantiated for a data persistence class, and contains generic methods that can return results of the instantiated type. A simple example of a generic method is to get an entity using its id as input.

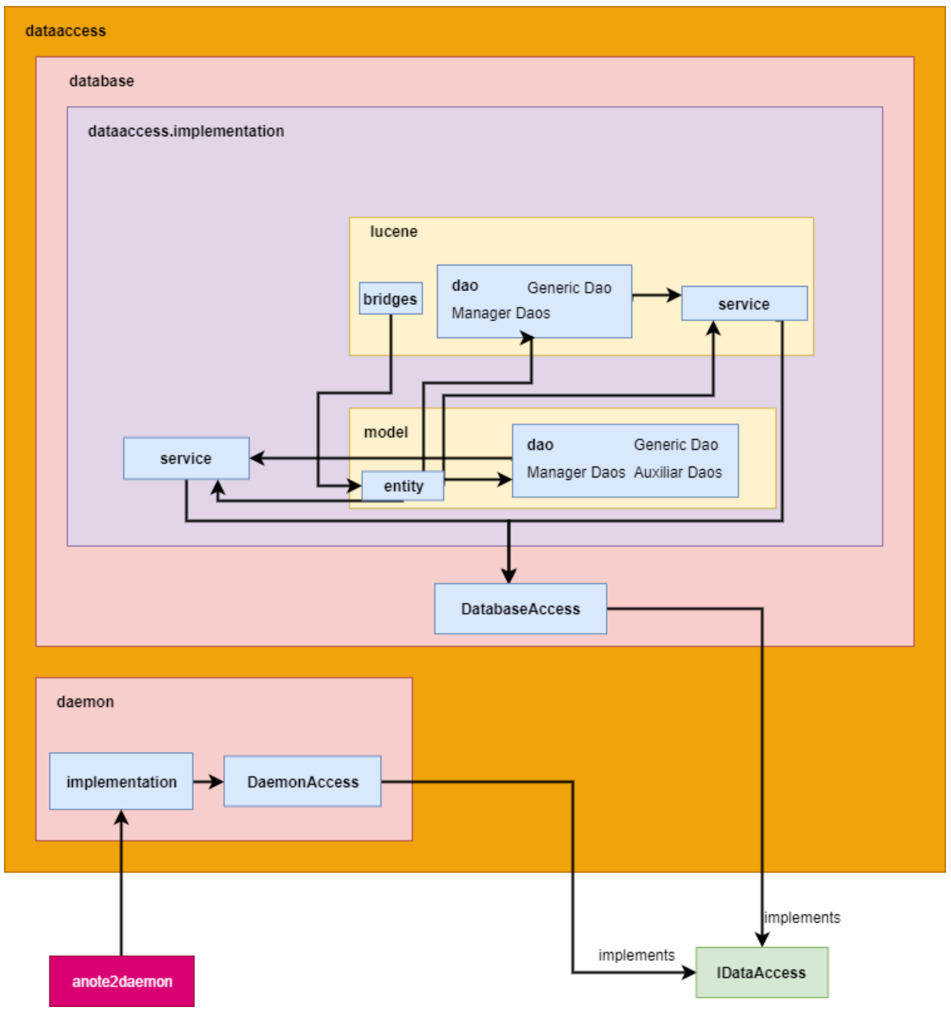


Figure 3.: *@Note* data access packages, entities and their relations

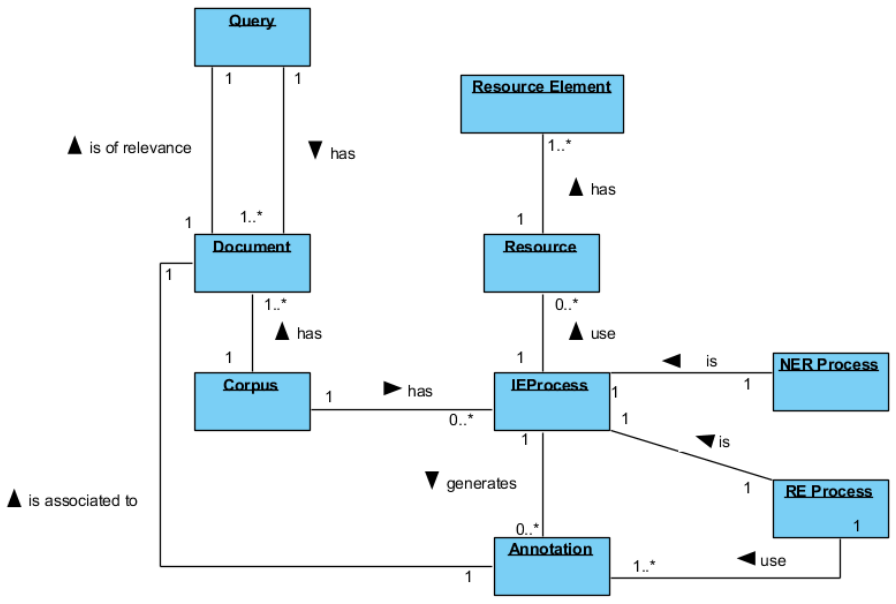


Figure 4.: *@Note* Domain model

3.2.3 *@Note domain model*

A domain model captures the entities of a problem and their relations. Figure 4 contains the most important entities to understand the system. The meaning of each entity is the following:

- ***Document***: *Documents* are unstructured texts with relevant metadata like authors, year of publications etc.
- ***Query***: *Query* is a search on a database for potential *documents* of interest related to a specific task. *Documents* associated to a *query* have relevance related to the *query*.
- ***Corpus***: This entity represents a structured set of *documents* over which *IE processes* extract information.
- ***Resource***: A *Resource* represents formally knowledge on a specific domain (e.g. biological entity or relation type). BTM processes can be defined with the option/necessity on the inclusion of resources to run.
- ***Resource Element***: This entity specifies a selected *term* representing an element of a specific domain with its synonymous and external databases.
- ***IEProcess***: *IE processes* extract information from the unstructured text on *documents* of a *corpus* by generating annotations.

- ***NERProcess***: This entity specifies *IE processes* of the NER type, which identify and annotate entities of interest on *documents*.

- ***REProcess***: This entity specifies *processes* of the RE type, which find relations involving entities previously annotated by a NER *process*.

Users and permissions

Users belong to one of two groups, Admin or General. Admins have access to administration functionalities such as: changing permissions and creating users.

The *@Note* platform has five major entities: *Queries*, *Corpus*, *IEProcesses*, *Resources* and *Documents*. With the exception for *Documents*, accessibility is restricted based on user permissions. A permission to a resource can be classified as: *owner*, *read write*, *read* and *none*. Permissions and user groups have a significant impact on the development of the application, as the majority of requests to the database need to obey the logic imposed by user permissions. The users' permissions are also used to "remove" information, for example if a dictionary is no longer useful to a user, "removing" that dictionary is changing the permission of that resource to none. The dictionary now is not returned on a *query* with the set of dictionaries associated to this user, while for other users using that resource nothing changes. If the dictionary is again of use to the user, an admin can change the permission again. Only elements that have errors and should never have any application are deleted by changing its validity parameter on the database.

Query

Queries are IR operations to search for *documents*. If the search is related to an external database, the *documents* are retrieved and stored. *Queries* have a set of parameters that help on the selection of relevant *documents*. The relevance of a *document* to a *query* can vary on levels from *None* to *Relevant*.

As shown in figure 5, the *QueryOriginType* gives information on how the query was executed. Processes on the *@Note* platform for *queries* have a specific name associated to it, for example a query to the *Pubmed* database has an origin of *PUBMED*. Properties are key-value pairs that provide additional information on a *query*, like date ranges on *publications*.

Corpus

A *corpus* is a set of *documents* created by an *IR process*, selected from a given source with a given purpose executing a *query*, over which *BTM processes* can be executed. In figure 6, it is visible that like on the *Query properties*, there are pairs to store additional information about the *corpus*, such as if they are full texts or abstracts.

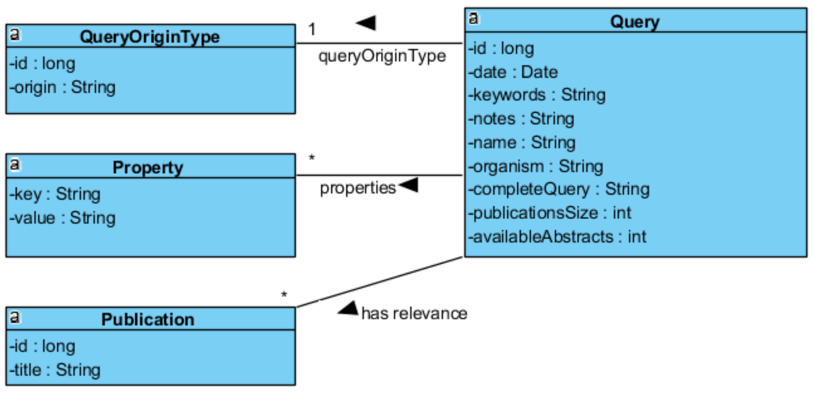


Figure 5.: *@Note Query* class diagram

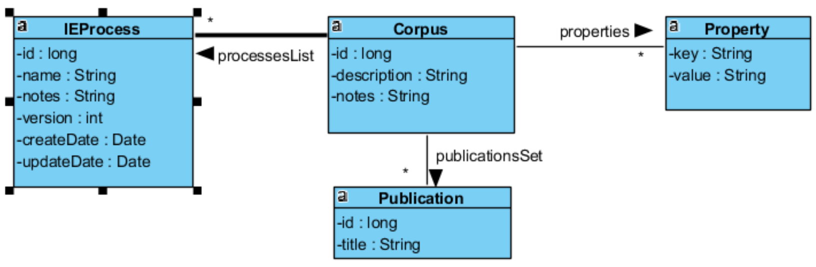


Figure 6.: *@Note Corpus* class diagram

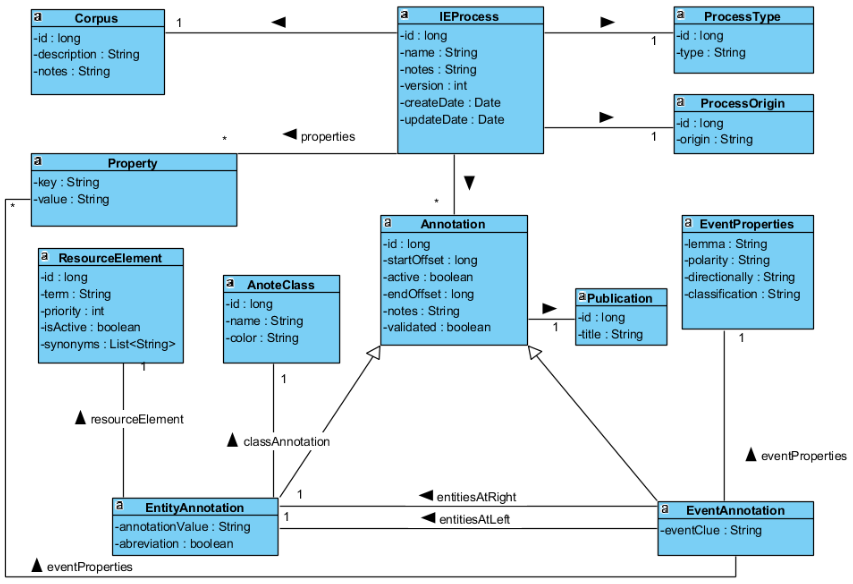


Figure 7.: *@Note IEProcess* class diagram

IEProcess

An *IEProcess* (Information extraction process) manages information related to a BTM process run. In figure 7, we have a diagram with the interactions of the entities involved. An *IEProcess* must be run over a *Corpus*, has a type (NER or RE), and have a reference to the *process* algorithm executed. If the *corpus' documents* have relevant information related to the executed process setup, it generates *annotations* of the *documents*. An *annotation* references a portion of text on a *document*, which can be identified by the start and end offsets. *Annotations* automatically generated from *IE processes* can later be curated by a user, that can validate them and add notes. *Annotations* can be related to entities or events on *documents*. Entities must reference a *resource element* used by the *process*, have a class that defines the color shown when presented on a *curator view*, and also indicate if the entity is an abbreviation. *Event annotations* reference the entities at its left and right and have an event properties object that defines additional information on the event.

Resource

Resources are auxiliary knowledge to *BTM processes* adding domain-specific knowledge. Each type of resource has a specific structure and complexity of relations between entities. However, they all have *terms* that represent a domain entity. *Resources* can also help the algorithm's precision, one example being stop words that identify segments of text that should be ignored even if they are candidates to generate an *annotation*. As it can be seen

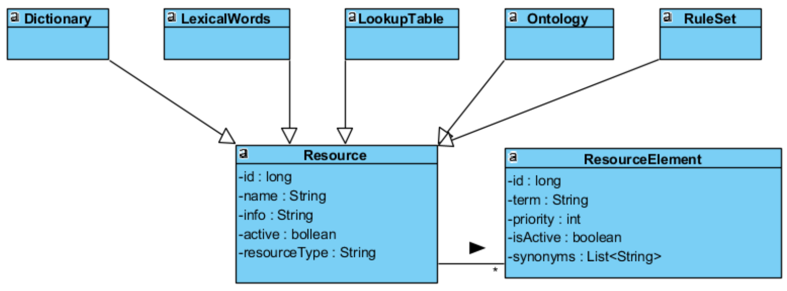


Figure 8.: *@Note Resource* class diagram

in the figure 8, *resources* can be of type *Dictionary*, *LexicalWords*, *LookupTable*, *Ontology* or *RuleSet*

3.3 CACHE

The *@Note* application can be applied to an extensive database with millions of *documents* and *terms*. As such, even a simple count operation on the database can take a couple of minutes to complete. In order to provide a smoother experience in the web application, a cache system was implemented. This cache implementation in the *anote2daemon* stores some values returned by chosen methods. When the method is called for the same input the stored value is returned.

On *Spring*, caching can be enabled by adding some *beans* to the application context file and by adding annotations on *controller* methods. The stored values in cache can be purged if conditions change, e.g. on a method to return the total *documents* in the database cache value should be purged if new *documents* are added. The implemented *Spring* cache does not store values persistently, all data is lost if the application is terminated or restarted.

By adding a *<cache:annotation-driven / >* followed by a *bean* with id *cacheManager* and the class *org.springframework.cache.support.SimpleCacheManager*, with some more group declarations we register a *cacheManager*, the first step to apply caching in the system. The cache group is now ready to receive *beans* declaring each cache. Cache annotations on methods are added on *daemon controllers*, and it is necessary to declare them in the cache group. Declared *controllers* were: *publicationsController*, *corpusController* and *resourceElementsController*. To enable cache on a method it is used the annotation *@Cacheable*, and to remove cached values the annotation *@CacheEvict*. Both need to receive as an entry parameter the name of the cache. If the method they are associated to has a variable as input, it is also passed

Cache	Associated method	Controller
countAllPublications	countAllPublications()	PublicationsController
countAllDistinctColumnValuesFromPublications	countAllDistinctColumnValuesFromPublications()	PublicationsController
resourceContent	getResourceContent()	ResourceElementsController
resourceClassContent	getResourceClassContent()	ResourceElementsController
corpusStatistics	getCorpusStatistics()	CorpusController

Table 5.: Caches in *anote2daemon controllers*, with the method they are associated to

Method	Controller	Evicted caches
createMultiplePublications()	PublicationsController	countAllPublications countAllDistinctColumnValuesFromPublications
updatePublication()	PublicationsController	countAllDistinctColumnValuesFromPublications
addResourceElements()	ResourceElementsController	resourceContent resourceClassContent
addResourceElementsWithoutValidation()	ResourceElementsController	resourceContent resourceClassContent
removeResourceClass()	ResourceElementsController	resourceContent resourceClassContent
addResourceElementSynonyms()	ResourceElementsController	resourceContent
updateCorpus()	CorpusController	corpusStatistics
updateCorpusStatus()	CorpusController	corpusStatistics
registerCorpusProcess()	CorpusController	corpusStatistics
addCorpusPublication()	CorpusController	corpusStatistics
inativateCorpus()	CorpusController	corpusStatistics

Table 6.: Methods in *anote2daemon controllers* that evicts caches and the caches they evict

as a key value (spr, a). Table 5 contain the implemented caches and table 6 contains the methods where they are evicted.

3.4 SEARCH

This section explains the implementation of the new searching features on the application's domain. It starts with the *Hibernate Search*, going through its database, DOM *queries* and *services*. It closes with Boolean expression search, two implementations of attribute grammars to search for *documents*, the first using words in text and the other *resource elements* annotated in the text.

3.4.1 *Hibernate Search*

Analysers

To facilitate/enable different types of queries, most fields are indexed using multiple analysers. For the custom analysers, we used two *tokenizer factories* [7]:

- *StandardTokenizerFactory*: This *tokenizer* splits the text field into *tokens*, treating whitespace and punctuation as delimiters including hyphens. Delimiter characters are discarded, with the following exceptions:
  - Periods (dots) that are not followed by whitespace are kept as part of the *token*, including Internet domain names.
  - The "@" character is among the set of *token*-splitting punctuation, so email addresses are not preserved as single *tokens*.
- *KeywordTokenizerFactory*: This *tokenizer* treats the entire text field as a single *token*.

The filters used are:

- *stopFilterFactory*: Removes words contained in a stop words list. The default list has english stop words list with words usually not useful to a search [8].
- *LowerCaseFilterFactory*: Lowercase all words.
- *EdgeNGramFilterFactory*: generates edge n-gram *tokens* of sizes within the given range.

The *KeywordsSplitter analyser* uses the *StandardTokenizerFactory* and the *filter StopFilterFactory*. This is a very general *analyser* only using the *StopFilterFactory* for some optimization.

The *toLowerCase analyser* is very identical to *KeywordsSplitter*, with an additional *filter* to convert all words to lowercase. *Queries* applied to data transformed by this *analyser* are non case-sensitive.

The *keywordEdgeAnalyzerCS* and *tokenEdgeAnalyzerCS analysers* generate *terms* applying an *EdgeNGramFilterFactory*. These *terms* are substrings of the token given as input, starting at value *minGramSize* with a maximum range of *maxGramSize*. For example, the *token* textmining generates the following terms tex, text, textm, ..., textmining. This method of storing data is useful for imprecise text search, when we want more results, even if less reliable they have an higher chance of containing what is looked for.

7 https://lucene.apache.org/solr/guide/6_6/tokenizers.html
8 https://lucene.apache.org/core/4_6_0/analyzers-common/org/apache/lucene/analysis/core/StopAnalyzer.html#ENGLISH_STOP_WORDS_SET

Search fields

To make a *field* searchable we need to apply some annotations configuring how the index is created. The tables 15, 14, 16, 17, 18, 19, 13, 20, 21 and 22 contain for each data persistence class the database column with the associated *Lucene @Fields* listed by name.

The *field* names listed follow a naming convention in their definition. All *field* names finished by ”CS” are indexed and analysed by *keywordsSplitter analyser*, the ones finished by ”NCS” use the *analyser toLowerCase* and those finished by ”Sort” are not analysed. In this last case, just the *field* text is stored to enable sorting by that *field* the results returned by a *query*. We can observe the application of these rules at data persistence class *Publications* for the column *pub_title*:

```
@Fields(value = {
    @Field(name="titleCS",index=Index.YES, analyze=Analyze.YES, analyzer =
        @Analyzer(definition="KeywordsSplitter"), store=Store.NO),
    @Field(name="titleNCS",index=Index.YES, analyze=Analyze.YES,analyzer =
        @Analyzer(definition="toLowerCase"), store=Store.NO),
    @Field(name = "pubTitleSort", analyze = Analyze.NO, store = Store.YES)
  })
@SortableField(forField = "pubTitleSort")
```

For *field* names starting by ”keywordEdgeNGram_”, the *field* is analysed by *keywordEdgeAnalyzer*, starting by tokenEdgeNGram_ it is used the *tokenEdgeAnalyzer*. We can observe this at *ResourceElements* data persistence class for the column *res_element*:

```
@Field(name = "keywordEdgeNGram\_res\_element", index = Index.YES, store =
    Store.NO, analyze = Analyze.YES, analyzer = @Analyzer(definition = "
    keywordEdgeAnalyzer"), boost = @Boost(2)),

@Field(name = "tokenEdgeNGra\_res\_element", index = Index.YES, store = Store.
    NO, analyze = Analyze.YES, analyzer = @Analyzer(definition = "
    tokenEdgeAnalyzer"))
```

Lucene bridges

All index *fields* of *Hibernate Search* must be converted into *Strings*, the majority of this work is done automatically by some built-in bridges. However, this does not cover all cases like class relations. To help *Hibernate Search* convert these relations to string based, we implemented two-way bridges, all with the same structure: initialization of a string with

Variable	Type	Description
value	String	Text search
fields	List<String >	List of fields to search at
wholeWords	Boolean	If true the query must use the text search as a continuous segment
keywords	boolean	If true the text search must be divided word by word
caseSensitive	Boolean	If true the query must be case-sensitive, otherwise, it must do a non case-sensitive search
restrictions	Map<String, String >	Restricts returned results of pairs field value. For example return just processes of a specific corpus
filters	Map<String, List<String >>	Filter results, allows for more than one value for the same field
expression	Boolean	The text search is a Boolean expression

Table 7.: SearchPropertiesImpl variables with description

a value used to "connect" string representations of the two classes the *bridge* is applied to. This "connection" is used by the two methods implemented in the *bridge*:

- *objectToString*: Converts the object into a string to be in the *Lucene* indexation.
- *stringToObject*: Converts a string generated by *objectToString* back to an object.

Query

All *Lucene* data access *services* have methods to query the *Lucene* database. We will explain how to query for paginated data, sorted by a field that returns results based on user permissions to that resource. These have the parameters *searchProperties* (of type *ISearchProperties*), *permission* (string), *index* and *paginationSize* (integers), *asc* (Boolean), and *sortBy* (string). The permission refers to the user requesting the data, required to only return what the user has access to; (*index* and *paginationSize* are used to return the intended page of results; if *asc* is true, the results are in ascending order, otherwise they are returned in descending order; *sortBy* indicates the *field* the results are sorted by. *SearchPropertiesImpl* class implements the search configurations the presentation layer uses to communicate with the business layer how to make the *query*. Table 7 has *SearchPropertiesImpl*, that implements *ISearchProperties*), variable names, their type and how they are used.

Lucene queries need the unique index field defined at *@Field* definition, these are stored at *enums* created for each searchable data persistence class. Using the *enum ProcessLuceneFields* created for the data persistence class *Processes*, we can define the following enum where for each database column is attributed a suggestive name, usually the names of the variables of the corresponding business class. The exception is the name *none* that is used to represent the absence of sorting by an entity:

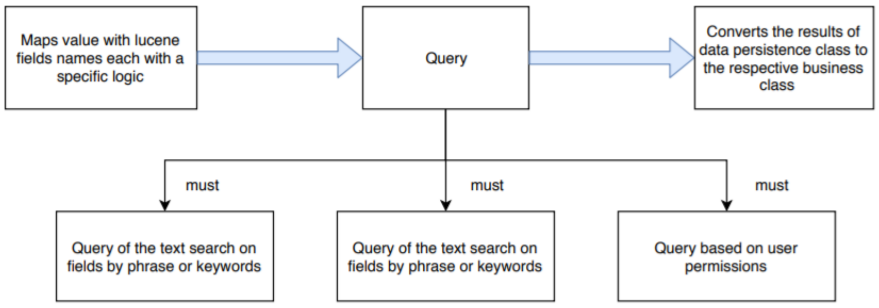

Maps value with lucene fields names each with a specific logic
Query
Converts the results of data persistence class to the respective business class
must
must
must
Query of the text search on fields by phrase or keywords
Query of the text search on fields by phrase or keywords
Query based on user permissions

Figure 9.: Search method in a Lucene Service with permissions and filters

```
processOrigin("processOrigins.pro_po_descriptionNCS", "processOrigins.
    pro_po_descriptionCS", "processOrigins.pro_po_descriptionSort",
    SortField.Type.STRING ),
processType("processTypes.pro_pt_typeNCS", "processTypes.pro_pt_typeCS", "
    processTypes.pro_pt_typeSort", SortField.Type.STRING ),
notes ("pro_notesNCS","pro_notesCS", "none", null),
name("pro_nameNCS","pro_nameCS","proNameSort",SortField.Type.STRING),
none("","","none",null);
private final String NCS;
private final String CS;
private final String SORT;
private final SortField.Type SORTTYPE;
```

Each entity name has a correspondence to a *Lucene* index field name. The exception is the parameter *SORTTYPE* that instead of having a *field* name, has a *SortField* type necessary for sorting. The enum then has methods to retrieve the associated name based on one of the four parameters.

In figure 9, we have a visual representation on how the search method is structured. First, the function inputs are structured as maps, each having its own role for sub-queries. When necessary, the respective enum is used to get the unique field name. The next step is to call the *query* defined at *GenericLuceneDaoImpl*, its results are then converted to the business equivalent.

3.4.2 *Boolean search*

One of the major tasks of the web application is to empower its users with powerful querying tools. Almost all the information is stored using *Lucene* indexation and has search methods by phrase and keyword implemented. However, the *document* database is very extensive and always growing, and so it was decided it required a more precise and configurable query method. A Boolean query accomplishes this by refining requests with logical operators.

The application provides two distinct Boolean query operations on documents:

- *Global search*: This Boolean search uses *resource elements*, more specifically, their unique identifiers, as arguments and queries annotated *documents* for the results. This type of search allows for example to combine proteins and genes logically to retrieve relevant documents.
- *Expression*: This Boolean search uses words or phrases as arguments to search on *Lucene*s index database for the results.

To accomplish the goal of Boolean querying for *documents*, we used the *Java Compiler Compiler* (*JavaCC*). *JavaCC* is a parser generator and lexical analyzer generator for *Java*.

Expression grammar

In the introduction to this chapter, we talked about the grammar to be applied for *documents*. However, it can be applied to any class that implements the *GenericLuceneDao* and has data indexed by *Lucene*. It is dependent on a *Lucene service*, and on a list of index *fields* to query for results. To write a Boolean expression able to combine words and phrases (sequence of words that should be combined on the search) with logical operators we defined the *tokens* given in table 8.

Figure 10 shows the following Boolean expression's grammar functions and their interactions:

- *one_line*: initializes an array of *queries* to be manipulated by the operation; when the expression reaches the terminal symbol, the expression reached its end and the list of *queries* must only have one element that is returned.
- *operation*: inherits a list of *queries* (*queriesIn*), and initializes a new list *queriesOut*. When the end of the operation is reached, it checks for three different cases: the operation entered the *andOp* path, in this case the *queries* in *queriesOut* are combined into a single *query* with the *AND* logic. This query is then added to *queriesIn*; The *orOp* path is very similar to the previously described only changing the logic by which the *queries* are

Token	Definition	Explanation
Word	([″&″, ″+″, ″—″,″!″, ″″̈,″ ″,″(″,″)″, ″;″])+	Represents words of the text search
AND	(″&″ — ″+″ — ″AND ″)	Characters that represent the logical operator and.
OR	(″—″ — ″OR ″)	Characters that represent the logical operator or.
NOT	(″!″)	Character that represent the logical operator not.

QUOTED	`"\""`
	`(`
	`"\\" ~[]`
	`|`
	`~["\""]`
	`)*`
	`"\""`
		Represents a phrase, the quotation marks, allows the content to be considered as a single entity.

	(	In a term implies that an operation will follow it.
	)	Indicates that an operation preceded by (has finished.
	;	End of the Boolean expression

Table 8.: *Tokens* and terminal characters of the Boolean expression

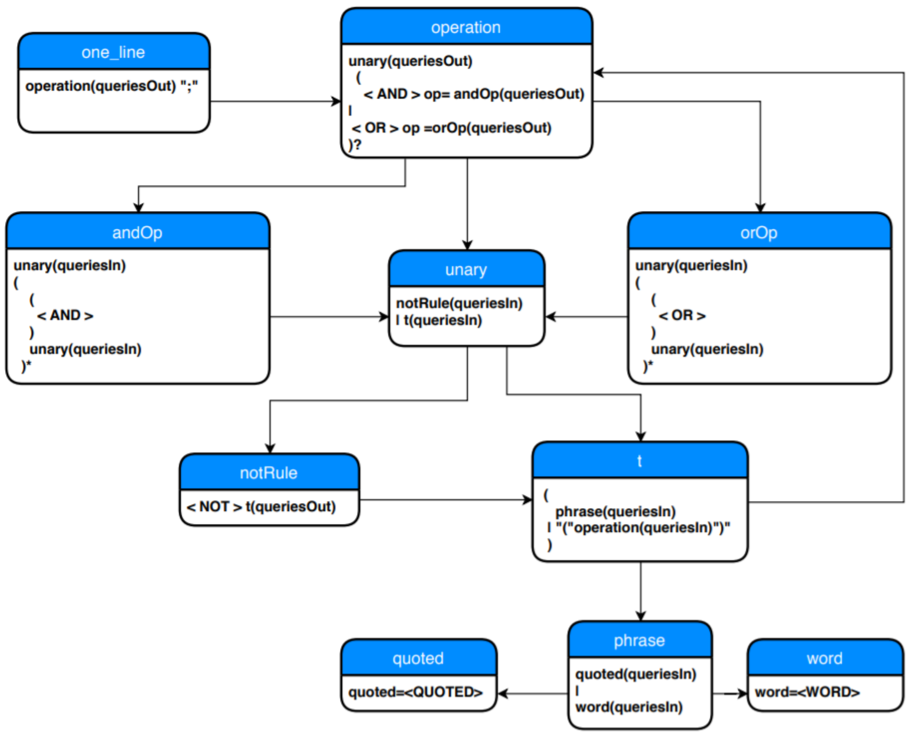


Figure 10.: Boolean expression grammar

combined to *OR*. The third option is if the operation only contains an unary, in this case no logic operation is applied to the queries at *queriesOut*, they are simply added to the *queriesIn* list.

- *notRule*: inherits a list of *queries*, *queriesIn*, and initializes a new list *queriesOut*. When it reaches the end, applies the logic of the *NOT* Boolean operator to combine the *queries* at *queriesOut*, it is then added to *queriesIn*.
- *word*: Generates a *query* using the string found by *WORD* and adds it to the list of *queries* inherited.
- *quoted*: Generates a *query* using the string found by *QUOTED* and adds it to the list of *queries* inherited.

Some examples of Boolean expressions are provided below (< >refers to the variable symbol):

- `<WORD>`
- `<WORD> AND <QUOTED>`
- `<WORD> AND <QUOTED> AND <WORD>`

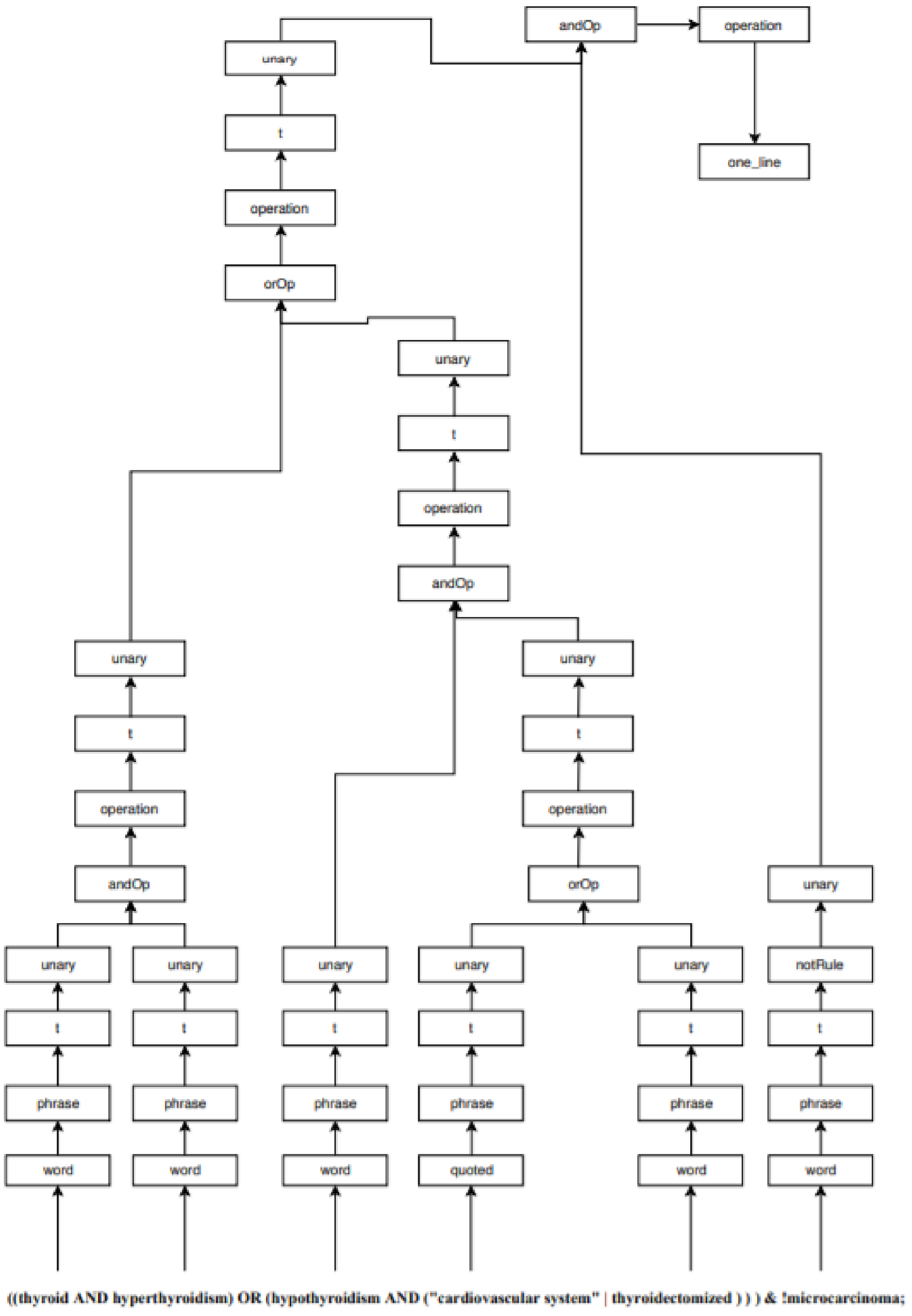


Figure 11.: Boolean expression example tree

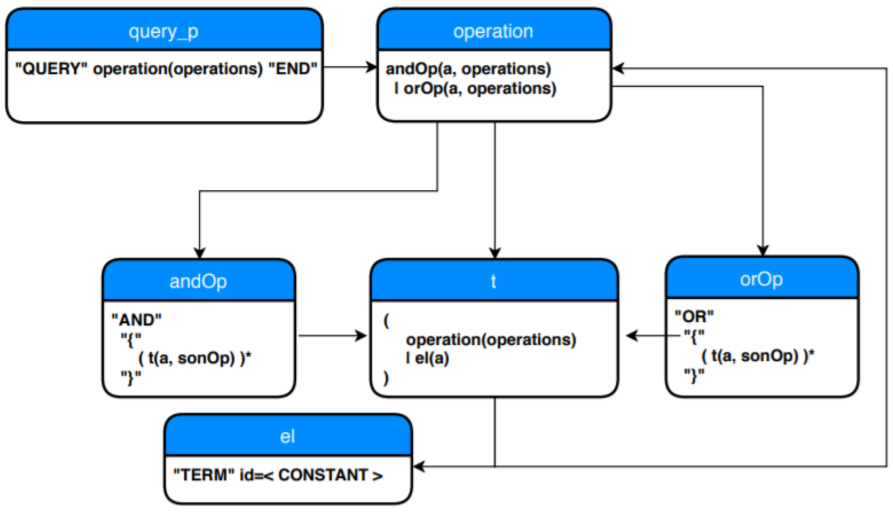


Figure 12.: Global search expression grammar

- `(<WORD> AND <WORD>) OR <QUOTED>`
- `<WORD> AND (<WORD> OR <QUOTED>)`
- `<WORD> AND !(<WORD> OR <QUOTED>)`

Global search

A Boolean search by the *global search* method must allow logical operations on BTM *resource elements*: the *annotationService* has the method to fetch *documents* based on the *resourceElement*, and as such it is a dependency to this grammar. To write a Boolean expression in this language it is necessary to provide the resource elements unique identifiers, which are captured using the token CONSTANT defined as:

```
<CONSTANT : (< DIGIT >)+ > | < #DIGIT : [ "0"-"9" ] >
```

The terminal symbols defined are: *QUERY*; *END*; *AND*; {; }; *OR*; *TERM* .
Figure 12 maps *global search* expression's grammar functions and their interactions:

- *query_p*: initializes a list to store sets of *resource element* ids (Long) and has the terminal symbols *QUERY* and *END* that start and finish the expression, respectively. When the end is reached, the list must have only one set of *documents* that are the result of the *query*.
- *operation*: operation just initializes a new Set of Long that is inherited by either *andOp* or *orOp*.

- *andOp/orOp*: Both fetch the *publications* with the *resource element* annotated on it and apply the respective logic.

3.5 PROCESSES

This section goes through the implementation of the data process status, used to track the progress of processes/jobs, and report the result. It also has a subsection on the new *corpus* creation processes, that include the new progress tracking and report system for processes.

3.5.1 *Process data status*

@Note has many different types of operations/processes, from executing a *query* to fetch *documents* from an external database to run an NER process. The system runs them in threads and their progress is printed by a method to the standard output stream. The desktop application uses a super implementation on this method to store the progress of the process in a variable, allowing the progress to be displayed. However, with the introduction of the web application, we decided it was necessary to implement a more robust and complete approach to the tracking and reporting on processes. The problems to solve can be summarized as:

- The web application needs to be able to access the progress of the process.
- If a process fails, this should be evident together with the reason why it failed.
- It should be possible for a user to track her/his processes that are running.
- An administrator should have a clear picture on the processes run /running on the server.

The solution started by adding a table to the relational database. Figure 13 shows a representation on *MySQL Workbench* of that table. Each field has the following purpose:

- ***dps_id***: Primary key, incremental.
- ***dps_data_object_id***: Id related to the result of the process. For example, a *corpus* creation process stores here the id of the *corpus* it creates.
- ***dps_type_resource***: Identifies the type of process, the options are: 'queries', 'resources', 'corpus', 'ner', 're'.
- ***dps_status***: Identifies in which state the process is at, the options are: 'start', 'running', 'finished', 'stop', 'error'.

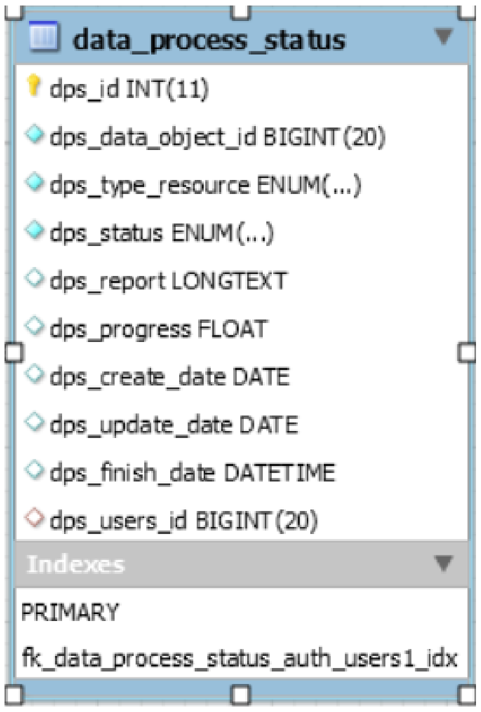

data_process_status
dps_id INT(11)
dps_data_object_id BIGINT(20)
dps_type_resource ENUM(...)
dps_status ENUM(...)
dps_report LONGTEXT
dps_progress FLOAT
dps_create_date DATE
dps_update_date DATE
dps_finish_date DATETIME
dps_users_id BIGINT(20)
Indexes
PRIMARY
fk_data_process_status_auth_users1_idx

Figure 13.: *@Note data_process_status* table of the relational database

- ***dps_report***: A textual report on the process.
- ***dps_progress***: A numerical value representative of the progress at the update time.
- ***dps_create_date***: Date of creation.
- ***dps_update_date***: Date of update.
- ***dps_finish_date***: Date of process completion.
- ***dps_users_id***: Foreign key, linking to auth_users.

Following the data access structure explained in the subsection 3.2.2, the next steps are to implement the DAO, *Service* and add its accessible methods to the *IDatabaseAccess* interface with the respective implementations. In the *anote2daemon*, it is also necessary to add the bean for the *process data status service* and a *controller*.

IDataProcessStatusAccess is the interface with data accessible methods related to *data process status*, extended by *IDataAccess*. A brief description on each method, or group of similar methods, follows:

- **addDataProcessStatus**: adds an element to the database, forces the *AuthUser* to be the session logged user.

- **updateDataProcessStatus** updates the entity, with the parameters in the object received as input. The exceptions are the creation date and *AuthUser* that should only be defined when the entity is created.
- **Get and count methods**: Get methods return one or more entities of type *IDataProcessStatus*; count methods return an integer, the number of results of the query. These methods have names correlated to the query, as follows:
  - **byId**: Find the entity using its id.
  - **User**: Find entities related to the logged user.
  - **Recent**: Entities are filtered by a *SIGDATERANGE* value applied to the date of update.
  - **Sorted**: Entities are sorted by id, in ascending order.
  - **OfType**: Filter the results by the type of process, types are defined at the enum *ProcessStatusResourceTypesEnum*.
  - **WLimit**: Only returns the first N results, the number is given as input.
  - **AdminPrivileges**: If the logged user has role admin, no further filter is applied. However if the user is not an admin only entities related to him are returned.
  - **Paginated**: With a pagination size and index returns the requested sub-list of results.

3.5.2 *Corpus creation*

The *anote2daemon* has an entry point responsible for running processes. Each process gets an unique identifier, which the entry point method uses to switch between the methods that execute in a thread the process. We added two additional methods for *corpus* creation, one for a *global search* query, and the other for a *Lucene* query on *publications* using *ISearchProperties* for its configuration.

The creation of a *corpus* has an auxiliary class (*CorpusCreateConfigurationImpl*), to store and manipulate data that can be fetched when it is required. As the new methods have additional information related to the search method, we implemented two classes extending *CorpusCreateConfigurationImpl*. Figure 14 is a class diagram representation exposing the relations, attributes and interface.

Figures 15 and 16 are sequence diagrams on the creation process of *corpus* from the entry point to run processes of *anote2daemon* for *global search* query and *lucene* query, respectively. They show how *DataProcessStatus* is used on each phase of the process to store its progress.

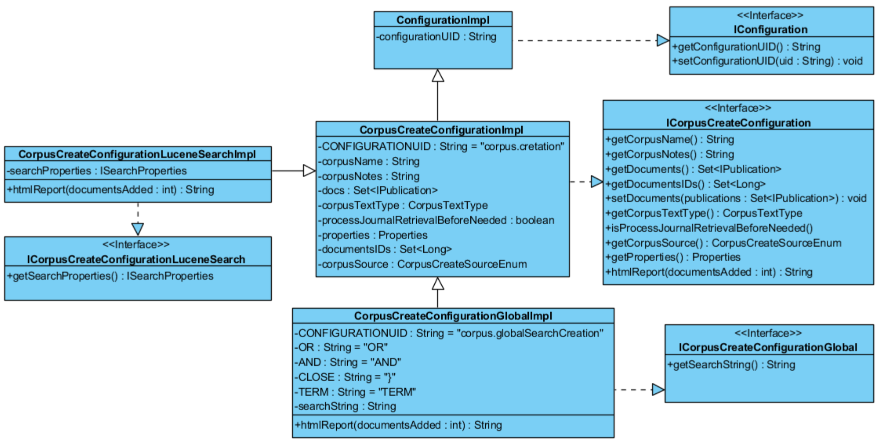

ConfigurationImpl
-configurationUID : String
<<Interface>>
IConfiguration
+getConfigurationUID() : String
+setConfigurationUID(uid : String) : void
CorpusCreateConfigurationImpl
-CONFIGURATIONUID : String = "corpus.cretation"
-corpusName : String
-corpusNotes : String
-docs : Set<IPublication>
-corpusTextType : CorpusTextType
-processJournalRetrievalBeforeNeeded : boolean
-properties : Properties
-documentsIDs : Set<Long>
-corpusSource : CorpusCreateSourceEnum
<<Interface>>
ICorpusCreateConfiguration
+getCorpusName() : String
+getCorpusNotes() : String
+getDocuments() : Set<IPublication>
+getDocumentsIDs() : Set<Long>
+setDocuments(publications : Set<IPublication>) : void
+getCorpusTextType() : CorpusTextType
+isProcessJournalRetrievalBeforeNeeded()
+getCorpusSource() : CorpusCreateSourceEnum
+getProperties() : Properties
+htmlReport(documentsAdded : int) : String
CorpusCreateConfigurationLuceneSearchImpl
-searchProperties : ISearchProperties
+htmlReport(documentsAdded : int) : String
<<Interface>>
ICorpusCreateConfigurationLuceneSearch
+getSearchProperties() : ISearchProperties
CorpusCreateConfigurationGlobalImpl
-CONFIGURATIONUID : String = "corpus.globalSearchCreation"
-OR : String = "OR"
-AND : String = "AND"
-CLOSE : String = ")"
-TERM : String = "TERM"
-searchString : String
+htmlReport(documentsAdded : int) : String
<<Interface>>
ICorpusCreateConfigurationGlobal
+getSearchString() : String

Figure 14.: *Corpus* configuration classes diagram

3.6 @NOTE WEB

The *@Note* web application is a tool with a friendly interface that allows an easy and fast access to information. It also contains some management features: to delete items, change permissions and see the state of running/finished jobs, among others.

3.6.1 *@Note web structure*

To create the application, the first step was to use the command, *ng new*, of the *@Note CLI* API. It is responsible for starting a project creating a folder structure with configuration files and the root *module*. Core *modules* for any *Angular* application are also added to the node *modules* folder, additional *modules* are added with the *npm* install command.

File structure

The *src* folder contains the main page, *index.html*, the global styles, the *assets* folder which holds configurations used by the application, such as images and translation files, and also, the *app* folder that contains the application's *modules*. Figure 17 is a tree representation of the web application's folder file structure for *modules* with pages.

Unlike the shared folder, other folders contain *modules* with pages associated to a specific context of the application. They all follow the structure presented in figure 18.

The main function of the *module.routes.ts* is to define the routes of the *module*. Each route is an object with: the name of the route; the *component* to show; the *service* that controls the access to the route, when needed (e.g. a route that can only be accessed by authenticated users, needs a *Service* to control the access). Another function of the *module.routes.ts* file

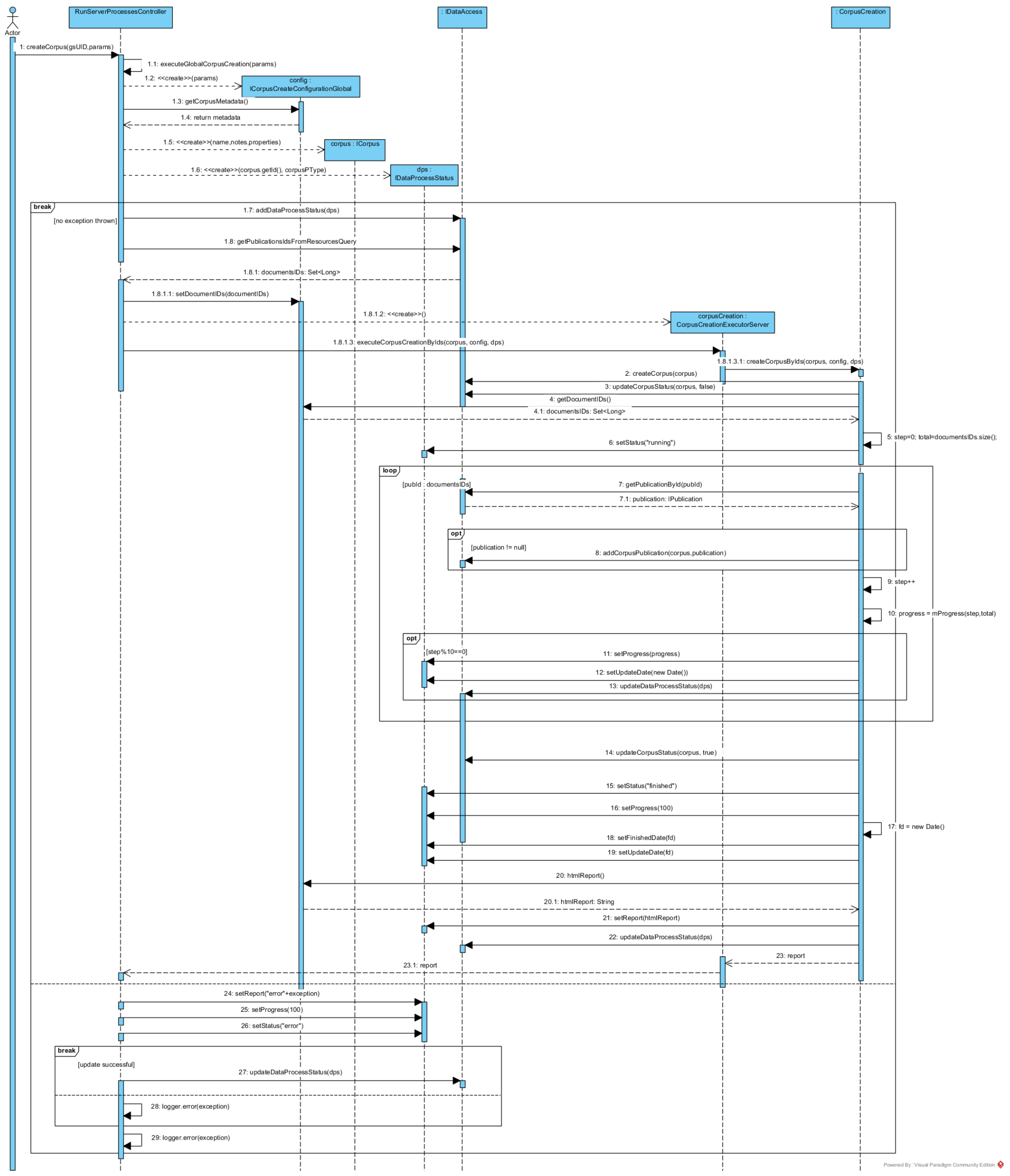

Actor
RunServerProcessesController
: IDataAccess
: CorpusCreation
1: createCorpus(gsUID,params)
1.1: executeGlobalCorpusCreation(params)
1.2: <<create>>(params)
config : ICorpusCreateConfigurationGlobal
1.3: getCorpusMetadata()
1.4: return metadata
1.5: <<create>>(name,notes,properties)
corpus : ICorpus
1.6: <<create>>(corpus.getId(), corpusPType)
dps : IDataProcessStatus
break
[no exception thrown]
1.7: addDataProcessStatus(dps)
1.8: getPublicationsIdsFromResourcesQuery
1.8.1: documentsIDs: Set<Long>
1.8.1.1: setDocumentIDs(documentIDs)
1.8.1.2: <<create>>()
corpusCreation : CorpusCreationExecutorServer
1.8.1.3: executeCorpusCreationByIds(corpus, config, dps)
1.8.1.3.1: createCorpusByIds(corpus, config, dps)
2: createCorpus(corpus)
3: updateCorpusStatus(corpus, false)
4: getDocumentIDs()
4.1: documentsIDs: Set<Long>
5: step=0; total=documentsIDs.size();
6: setStatus("running")
loop
[pubId : documentsIDs]
7: getPublicationById(pubId)
7.1: publication: IPublication
opt
[publication != null]
8: addCorpusPublication(corpus,publication)
9: step++
10: progress = mProgress(step,total)
opt
[step%10==0]
11: setProgress(progress)
12: setUpdateDate(new Date())
13: updateDataProcessStatus(dps)
14: updateCorpusStatus(corpus, true)
15: setStatus("finished")
16: setProgress(100)
17: fd = new Date()
18: setFinishedDate(fd)
19: setUpdateDate(fd)
20: htmlReport()
20.1: htmlReport: String
21: setReport(htmlReport)
22: updateDataProcessStatus(dps)
23: report
23.1: report
24: setReport("error"+exception)
25: setProgress(100)
26: setStatus("error")
break
[update successful]
27: updateDataProcessStatus(dps)
28: logger.error(exception)
29: logger.error(exception)
Powered By: Visual Paradigm Community Edition

Figure 15.: Create *corpus* from *global search* query sequence diagram

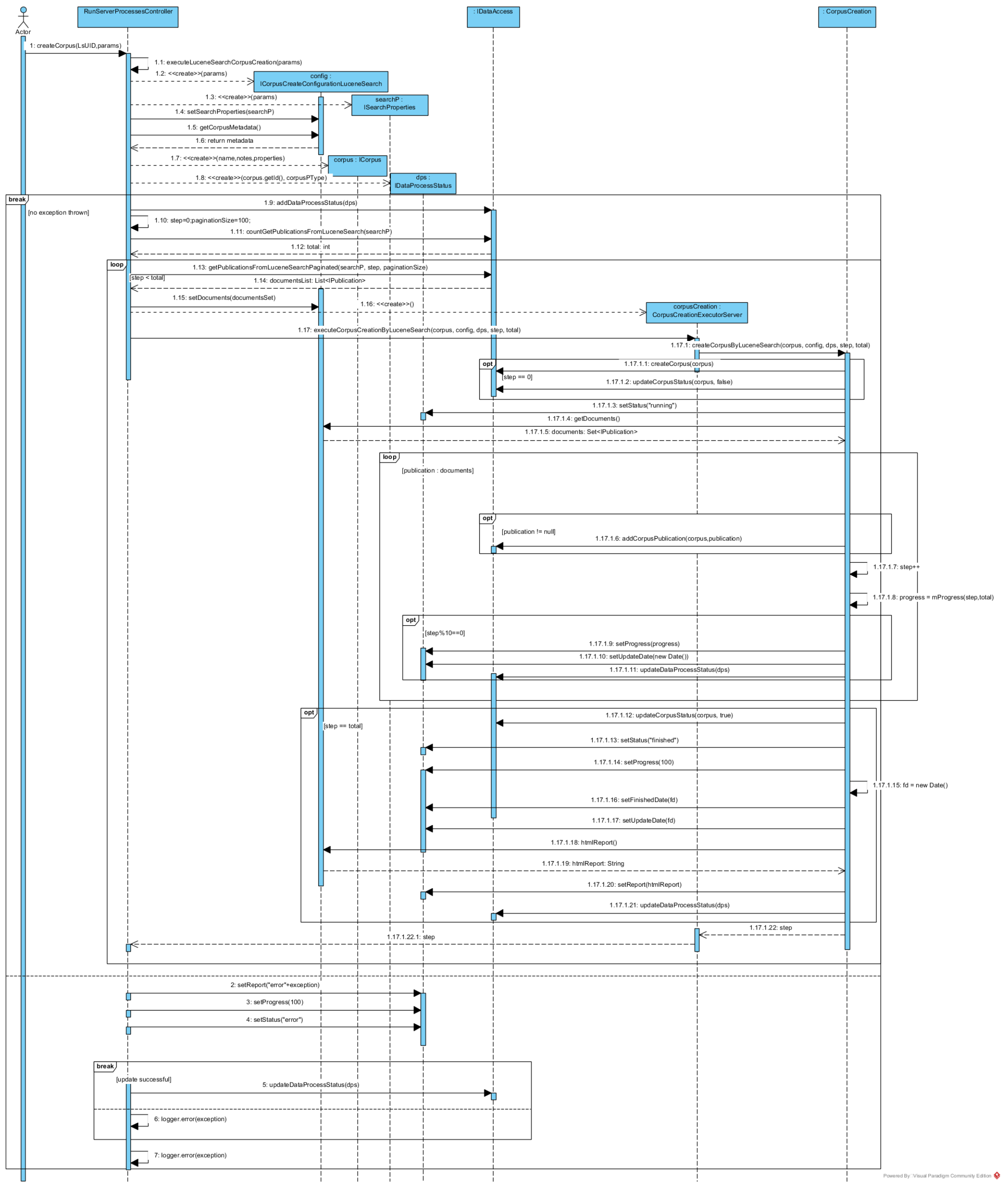

Actor
RunServerProcessesController
: IDataAccess
: CorpusCreation
1: createCorpus(LsUID,params)
1.1: executeLuceneSearchCorpusCreation(params)
1.2: <<create>>(params)
config : ICorpusCreateConfigurationLuceneSearch
1.3: <<create>>(params)
searchP : ISearchProperties
1.4: setSearchProperties(searchP)
1.5: getCorpusMetadata()
1.6: return metadata
1.7: <<create>>(name,notes,properties)
corpus : ICorpus
1.8: <<create>>(corpus.getId(), corpusPType)
dps : IDataProcessStatus
break
[no exception thrown]
1.9: addDataProcessStatus(dps)
1.10: step=0;paginationSize=100;
1.11: countGetPublicationsFromLuceneSearch(searchP)
1.12: total: int
loop
1.13: getPublicationsFromLuceneSearchPaginated(searchP, step, paginationSize)
[step < total]
1.14: documentsList: List<IPublication>
1.15: setDocuments(documentsSet)
1.16: <<create>>()
corpusCreation : CorpusCreationExecutorServer
1.17: executeCorpusCreationByLuceneSearch(corpus, config, dps, step, total)
1.17.1: createCorpusByLuceneSearch(corpus, config, dps, step, total)
opt
1.17.1.1: createCorpus(corpus)
[step == 0]
1.17.1.2: updateCorpusStatus(corpus, false)
1.17.1.3: setStatus("running")
1.17.1.4: getDocuments()
1.17.1.5: documents: Set<IPublication>
loop
[publication : documents]
opt
[publication != null]
1.17.1.6: addCorpusPublication(corpus,publication)
1.17.1.7: step++
1.17.1.8: progress = mProgress(step,total)
opt
[step%10==0]
1.17.1.9: setProgress(progress)
1.17.1.10: setUpdateDate(new Date())
1.17.1.11: updateDataProcessStatus(dps)
opt
[step == total]
1.17.1.12: updateCorpusStatus(corpus, true)
1.17.1.13: setStatus("finished")
1.17.1.14: setProgress(100)
1.17.1.15: fd = new Date()
1.17.1.16: setFinishedDate(fd)
1.17.1.17: setUpdateDate(fd)
1.17.1.18: htmlReport()
1.17.1.19: htmlReport: String
1.17.1.20: setReport(htmlReport)
1.17.1.21: updateDataProcessStatus(dps)
1.17.1.22: step
1.17.1.22.1: step
2: setReport("error"+exception)
3: setProgress(100)
4: setStatus("error")
break
[update successful]
5: updateDataProcessStatus(dps)
6: logger.error(exception)
7: logger.error(exception)
Powered By Visual Paradigm Community Edition

Figure 16.: Create *corpus* from a *publications lucene* query sequence diagram

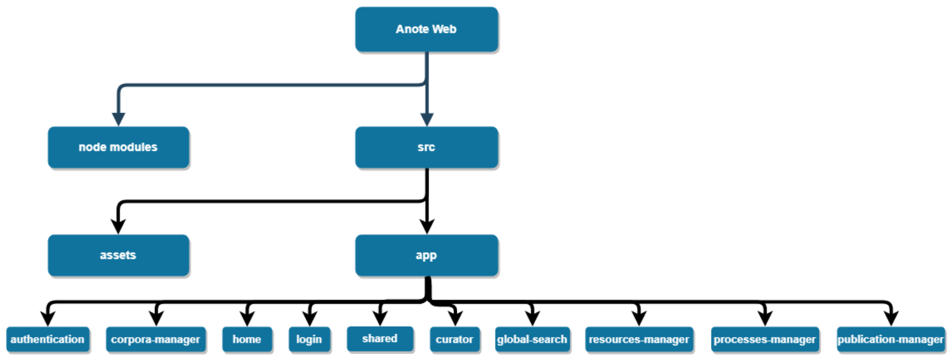


Figure 17.: *@Note web* file structure

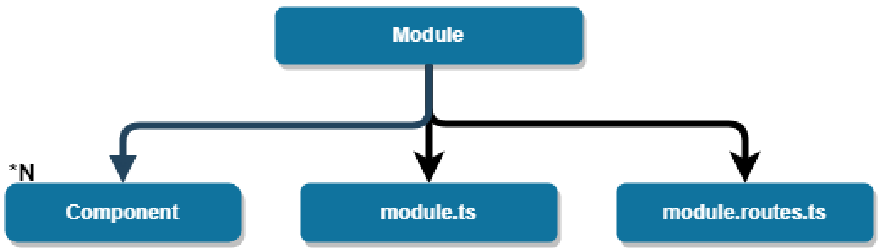


Figure 18.: *@Note web module* main structure

allows to store *components* associated with the *model* in two separate lists: *module components* and shared *components*. The *components* in the *module components* list are declared by the *module*. The *components* in the shared *components* list are imported by the *model*. The *shared* folder contains *Services*, *pipes*, *directives* and *modules* with no routes defined, their purpose being to be incorporated in other *modules*.

Configuration files

The *@Note* web application has two types of information read from files, in the *assets* folder:

- Configurations for the application that can change from build to build.
- Language files with translations on specific elements of the application.

Configuration files define the following parameters:

- *baseURL*: the path to the *anote2daemon* server.
- *mainLoader*: the name of the main loader type of the application.
- *singleLoader*: the name of the loader usually smaller, used in specific content.
- *processLoader*: the name of the loader type used in the processes running.

- *resourcesTypes*: the names of the resources present in the *@Note* database.
- *permissionIcon*: the path to the icon representing the permissions configuration button.
- *defaultAvatarIcon*: the path to the icon of the default avatar given to all users when none is present in the database.
- *version*: the current version of the web application.

A language configuration file with translations to a language has a JSON object, the keys are all upper case, its value can be a string representing a translation or another object following the same structure.

3.6.2 *Data access*

All the information related to the BTM domain of the *@Note web angular* implementation is requested to an *anote2daemon* web service. The methods that make requests are defined in *Services* mirroring the *anote2daemon controlers*. For example, in the *anote2daemon controller* package it contains a package called *publications* with a *PublicationsController* and a *QueriesController*, each defining the respective domain entry points. The angular application has in the *services* a mirror folder that must contain a *PublicationsService* and a *QueriesService Services* that implement the methods to access entry points implemented at their "mirror" *controller*. In this communication, the format used is JSON, and it is taken into consideration when parsing the response. The types of the object received are defined as *interfaces*, which are also a "mirror", in this case of the *Java* objects defined at *Core* and returned by the *anote2daemon* entry points. *Components* then use this *Services* to populate their *views* and request operations like the change of a permission or to create a *corpus*.

Authentication

The authentication in the *@Note web* application is not just a simple exchange, where the credentials are sent. The *anote2daemon* has security measures that ensure the messages following the login belong to the authenticated user while the session is active. The first security measure is to exchange a *token JSESSIONID* used on communications for that session. This is defined at *spring-security.xml*. This measure is simple to comply for the web implementation, if in the requests the parameter *withCredentials:true* is received, *Angular* automatically stores the session identity and/or sends the one it has stored in the message.

The second security measure is a *Cross-origin resource sharing (CORS)* implementation. This prohibits *Asynchronous JavaScript And XML (AJAX)* calls to resources residing outside the current origin. In a scenario where a person is authenticated in their bank's website, with CORS security, scripts from a website open in another tab should not be able to make

AJAX to the bank's API. CORS is a *World Wide Web Consortium (W3C)* specification implemented by most browsers that allows to specify in a flexible way what kind of cross domain requests are authorized (spr, b). To enable requests from the web application to the *anote2daemon*, we implemented a *CorsFilter* class where all the domains allowed to request are defined. This class is used in the CORS filter security declaration at *web.xml*. In the *Angular* implementation, it is required to provide in the application main module a *CookieXSRFStrategy* with the *cookie* and the *cross-site request forgery* header names defined at the *anote2daemon*.

3.6.3 *Modules*

This subsection contains the *modules* implemented, starting with a simple overview and explaining in detail the *modules* with higher relevance.

Models overview

As stated in section 3.6.1, *models* can either have routes to pages in the application or be defined at the shared folder, to have their *components* reused by multiple *models*. To better understand the scope of the application, the following enumeration contains the *modules* that incorporate pages with a succinct explanation for each page:

- Manager *modules* have pages related to an entity:
  - *corpora-manager*: has a page to list the *Corpora* available to a user, a page for a single *corpus* and a page with information on a *IEProcess* of a *Corpus*.
  - *processes-manager*: has a single a page displaying all the available *IEProcesses* to the user. Since an *IEProcess* must be run on a *corpus*, the page for a single *IEProcess* is handled by the *corpora-manager*.
  - *publication-manager*: has pages directly related to *documents* and pages of *queries* that search for *documents*. It has the page with all *documents*, the single *document view* page, all queries available to the user page and the page for a single *query*.
  - *resources-manager*: has two pages, one for all *resources* available to the user and the other for a single *resource*.
- *authentication*: This *module* has the pages responsible for authenticating the user, the page for user details edition and administration.
- *curator*: This *module* has the page to see a *document* annotated by at least one *IEProcess*, allowing the user to visualize those *annotations* on the *document*, and also a page with a form with *document* identifier and *IEProcess* identifier to reach the *curator* page for that *document* with the *annotations* for that process already visible.

- *global-search*: This *module* has a page responsible for the search operation that allows the user to find *resource elements* and manipulate them to build a Boolean expression to query annotated *documents*. The other two pages are related to the results of the query.
- *home*: Module with some informative pages for the application. It contains the *home* page and the *contacts* page.

The shared *modules* enumeration contains a succinct explanation on the *module* and/or the reusable *components* they contain:

- ***corpora***:
  - *corpus-notes*: responsible for displaying notes associated to a *corpus*; if the user has permission, enables the edition.
  - *create-corpus*: has a form with metadata for the *corpus* and methods for each type of *corpus creation* submit operation.
- ***data-process-status***:
  - *data-process-status-table*: table with the *data process status* processes the user has access.
  - *running-process-status*: *data process status* with some visualization options; periodically updates the state on the status of running processes.
  - *show-data-report*: displays the report on a *data process status*.
- ***footer***: has a *component* with the application's footer.
- ***navbar***: *module* of the application navigation bar.
- ***privileges***: *module* for *components* related to changing permissions. Contains a *component* of a table that can be applied for multiple entities to change permissions for users on that entity.
- ***processes*** : has a *component* displaying details of a *IE process*.
- ***publications***:
  - *publication-informations*: displays information on a *document* like meta-data and notes. The latter can also be edited.
  - *publications-table*: table for *documents* reusable with different applications. Example: the *documents'* table for *documents* on a *query* shows the *document* relevance to the *query*; *Documents'* table for *corpus* must only show the *documents* associated to that *corpus*.

  - *query-relevance*: display of *document* relevance to a *query* with the ability to change the relevance.

- ***queries***: has a *component* that displays details of a *query*.
- ***resources***: has two *components* that display details, one for a *resource* and the other for a *resource element*.
- ***search***:
  - *search*: *component* with the several search options like text search and filtering results. Knows how to make the request and returns the results. It is applied on tables to enable search.
  - *show-marked*: component that highlights text based on a search.
- ***table***: *module* responsible for displaying data in a table used in many of the other *components* of the application. Some operations of the table are: pagination, expandable rows, select visible columns, text search provided by a *search component* integration and sort by column.

Data tables

The focus of the current state of the *@Note web* application is primarily focused on visualization of information. Like the desktop application, many contents use paginated tables with search options. Tables on the web application use the *DataTableModule module*, which was first imported from *npm*, the world's largest software registry that developers can use *npm* to share packages (npm). It was integrated as an application *module*, to enable the addition of new features.

Input	Type	Description
ITEMS	any	Content displayed on the table rows, it is dynamic changing based on table operations.
ITEMCOUNT	number	Number of rows displayed on the table.
HEADERTITLE	string	Title of the table.
HEADER	boolean	Defines if the table shows a title.
PAGINATION	boolean	Defines if the table has pagination operations.
INDEXCOLUMN	boolean	Defines if the table has a column where each row shows its cardinality.

INDEXCOLUMNHEADER	string	The text shown at the index column header.
SELECTCOLUMN	boolean	Defines if the table has a column where each row has a check box
MULTISELECT	boolean	Defines if the select column header has a check box
EXPANDABLEROWS	boolean	Defines if the table has rows that can be expanded to show more information.
EXPANDABLEROWSHEADER	string	The text shown at the expandable rows header
EXPANDABLEWIDTH	number—string	Defines the width of the expandable column.
TRANSLATIONS	DataTable - Translations	Translation for elements of the table like pagination.
SELECTONROWCLICK	boolean	Defines if the row is selected by clicking on it.
AUTORELOAD	boolean	Defines if the items are reloaded when the table component starts.
EXPANDCOLUMNVISIBLE	boolean	Defines if the table expandable column starts as visible
SERVICE	string	Service to be used on the search component embedded on the table.
SHOWSEARCH	boolean	Defines if the table has search operation.
BLOCKLIMIT	boolean	Defines if the number of rows can be altered.
STARTLIMIT	number	Number of maximum starting rows.
TABLEPARAMS	*TableParams*	Defines the column by which elements are sorted by, if it is ascending or descending order, the offset and the maximum number of displayed rows.
INITIALSEARCHPROPERTIES	SearchProperties	A search to start the table on.
SORTBY	string	Defines the column where elements are sorted by.

Input	Type	Description
header	string	Text on the column header.
sortable	boolean	Defines if the column is sortable.
resizable	boolean	Defines if the size of the column can be altered.
property	string	Defines the property the column is associated to, of the items object.
styleClass	string	The class of the cells.
cellColors	*CellCallback*	Colors of the associated cells.
width	number — string	The width of the column.
visible	boolean	Defines if the column starts visible.

Table 10.: DataTable Columns Input

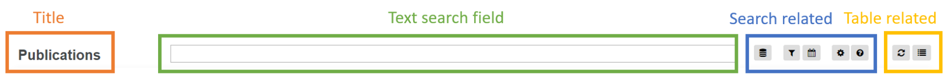


Figure 19.: DataTable Header of publications

SORTASC	boolean	Defines if it is ascending or descending order.
OFFSET	number	Defines the offset.
LIMIT	number	Defines the maximum number of displayed rows.

Table 9.: DataTable inputs

A data table is divided into Header, Columns, Rows and pagination, each with their *component*, *template* (*view*) and *style*. However, the data table *module* also contains a main *component* (*DataTable*) (table 9 contains the inputs available for this *component*). There are many variables that control different aspects of the table that can be manipulated, allowing for plentiful customization. Many elements that integrate the table are optional, however it is also possible to have a well-defined starting state of the table when first loading a page. Columns' *component* can also be tailored for the situation, the input variables in the table 10 explain what properties can be manipulated.

The header can hold the title of the table, text search field, search related buttons and table related buttons. Figure 19 shows, for the header of the *publications*' table, the four groups of *components*. The title gives a textual representation on the contents of the table; the text search field allows free text input; the search related and table related buttons have suggestive icons for their function. Table 11 has a description for each button.

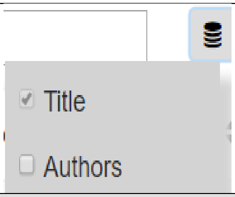
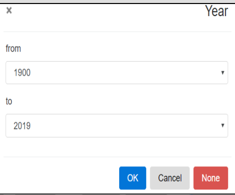
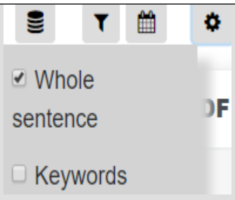
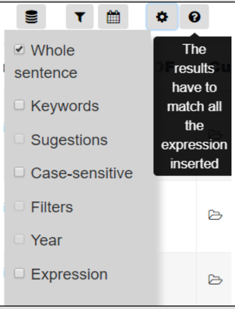
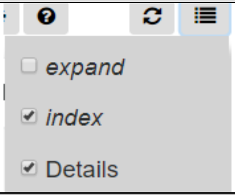
Button	Pressed	Description
🛢	 ☑ Title ☐ Authors	When pressed shows a list of selectable fields, at least one field selected is mandatory. The search will be on the fields selected.
▼		When pressed opens a modal with filters that can be added to the search.
📅	 × Year from 1900 to 2019 OK Cancel None	When pressed opens a modal with a date range selector.
⚙	 ☑ Whole sentence ☐ Keywords	When pressed shows a list of configurations that can be selected, applicable to the search. The possible configurations are: Whole sentence, Keywords, Case-sensitive, Filters, Year, Suggestions and Expression. Whole sentence, Keywords and Expression define how the text given as input on the text field is processed.
❓	 ☑ Whole sentence ☐ Keywords ☐ Sugestions ☐ Case-sensitive ☐ Filters ☐ Year ☐ Expression The results have to match all the expression inserted	When hovered offers a small hint of how the text search will be processed based on the selected option of the configurations button.
🔄		Refreshes the table
☰	 ☐ expand ☑ index ☑ Details	When pressed shows list of columns, the selected columns are shown on the table.

Table 11.: DataTable Header buttons

Table fields that the search can be based on use the *component show-marked* to highlight words/sentence corresponding to a search. The component has as input the text of the field, the object *searchProperties* that contains search related properties, like text search and configurations, and finally the name of the field. The *component* has a method based on regular expressions to find matching expressions to be highlighted. The algorithm takes into consideration the search options selected.

For example, on the input text "Norisoprenoids from Ulva lactuca." with the text search field with "ulva lactuca", there are some variations on search options, underlining what is highlighted:

- Only Whole sentence selected: Norisoprenoids from Ulva lactuca.
- Only Keywords selected: Norisoprenoids from Ulva lactuca.

Columns headers that can be clicked to sort results by the field they represent have a visual indication. This visual indication is a symbol comprised by one or two caret icons at the right of the column header. Clicking on it will sort table elements based on the chosen column. Two carets, one faced up on top of another faced down means that the table isnt sorted based on that column, clicking on it will sort the table by descending order represented by the caret faced down icon, clicking on it again will sort by ascending order and is represented by the caret faced up icon.

Rows can have the option to be expanded by double click, which will create a space between the clicked row and the row below containing more information. On both the cardinality and expandable column there is a caret icon that is faced right, if the row can be expanded but is collapsed; when it is expanded the icon changes to caret faced down. The pagination of data tables is located on the bottom of the table, the layout is shown in figure 20. From left to right, the list explains what is displayed and the user input options:

- Results: Elements currently on the table and the total of elements.
- Limit: Maximum number of elements displayed, this number can be edited by focusing the current number and typing a new number, using the up and down arrows on the keyboard or using the caret icons.
- Page navigation: Current page and last page; can be changed by focusing the current page number and typing the desired page or by using the intuitive buttons on both sides of this number.

Curator

A curator in BTM is an environment with a biomedical text with annotations clearly identified on the text. It should allow to manually curate the document and validate entities

Figure 20.: DataTable pagination

annotated by BTM processes. To give context to generated annotations, it is necessary to show some information on the BTM process and the resources it used. When a document is annotated by multiple processes the view should also link the annotation to the process.

The current curator of *@Note* web doesn't allow manipulation of *annotations* in the *document* or has visual relation cues between entities. The problem addressed is, for a *document* annotated by N *IE processes*, how to clearly identify *annotations* of each *process* and the context of that *annotation*.

The first consideration to solve this problem is that *documents* have an abstract and if available the full text and only one of them could contain annotations. We decided the solution would be to have a separation by these types of texts, allowing the user to change between an *abstract* and *full text view*.

To build the *curator*, it requires from the server:

- ***Publication***:
  - Metadata to give context on the *document* (e.g. the title).
  - The abstract to show annotations on the *abstract view*.
- The full text to show *annotations* on the *full text view*.
- ***IEProcess***:
  - Text type of the process to know if it annotates the abstract or full text.
  - Metadata of the *process* (e.g. the name of the algorithm).
- ***Resource***:
  - Metadata of the *resource* like its name.
  - The type of *resource*, example: dictionary.
- ***ResourceElement***:
  - The *term*, the name used to represent the element.
  - *Synonyms*.
  - The external identifiers of the element.
- ***AnoteClass***:
  - The name of the *class*.
  - The color that represents the *class*.

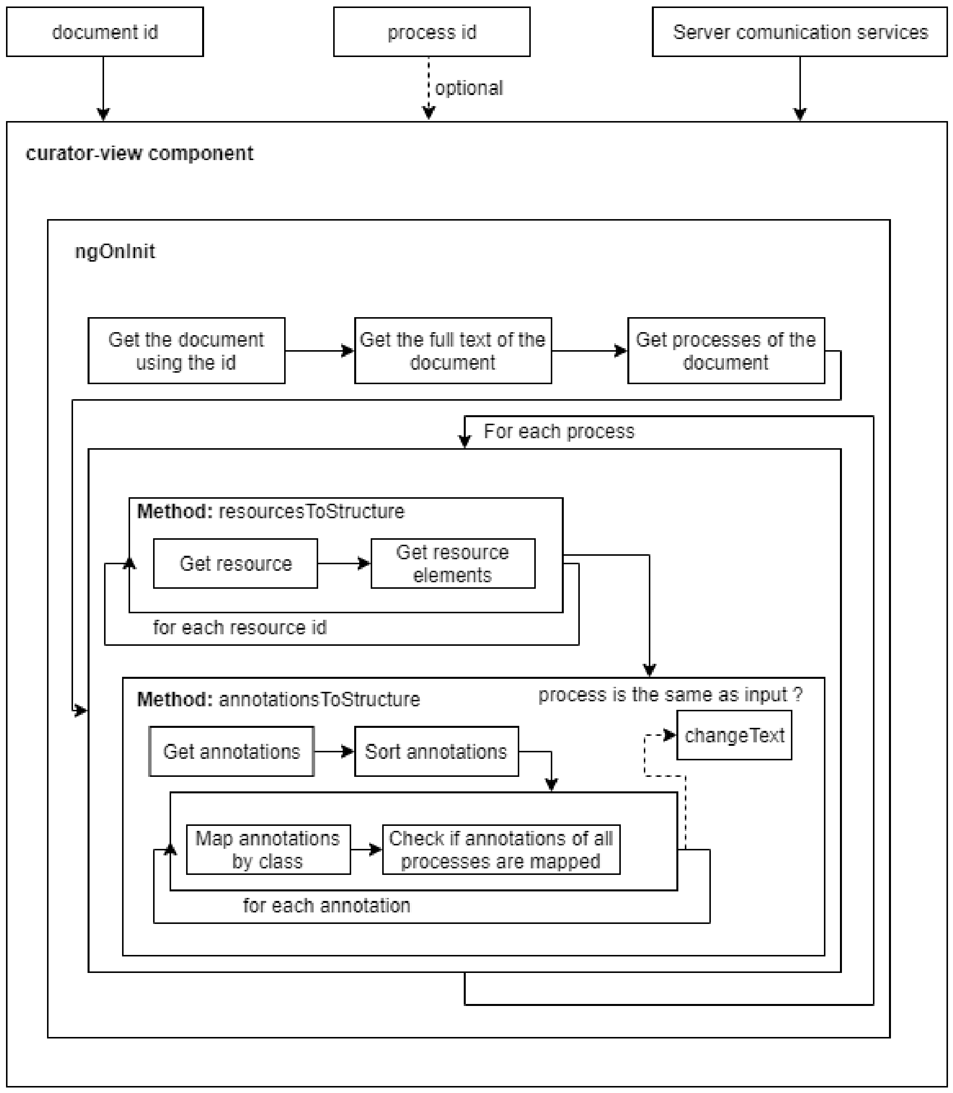

document id
process id
Server comunication services
optional
curator-view component
ngOnInit
Get the document using the id
Get the full text of the document
Get processes of the document
For each process
Method: resourcesToStructure
Get resource
Get resource elements
for each resource id
Method: annotationsToStructure
process is the same as input ?
Get annotations
Sort annotations
changeText
Map annotations by class
Check if annotations of all processes are mapped
for each annotation

Figure 21.: Method *ngOnInit* of *curatorViewComponent* component

- ***EntityAnnotation***:
  - *Annotation* value.
  - Start and end offsets to locate the annotated entity on the text.

Figure 21 goes through the most relevant steps of the *ngOnInit* method of *curatorView-Component component*. It is in this method that the majority of the information is requested and stored in structures that can be shared with the view to allow the display of information and control over annotated entities shown on the text. These structures are filled as the information is requested, for example when each *IE process* is iterated, and they are associated with a color. If the component has a *process* identifier as input, the function *changeText* is called to show the entities of that *IE process* on the text.

The process of showing annotated entities on text can be described with the steps:

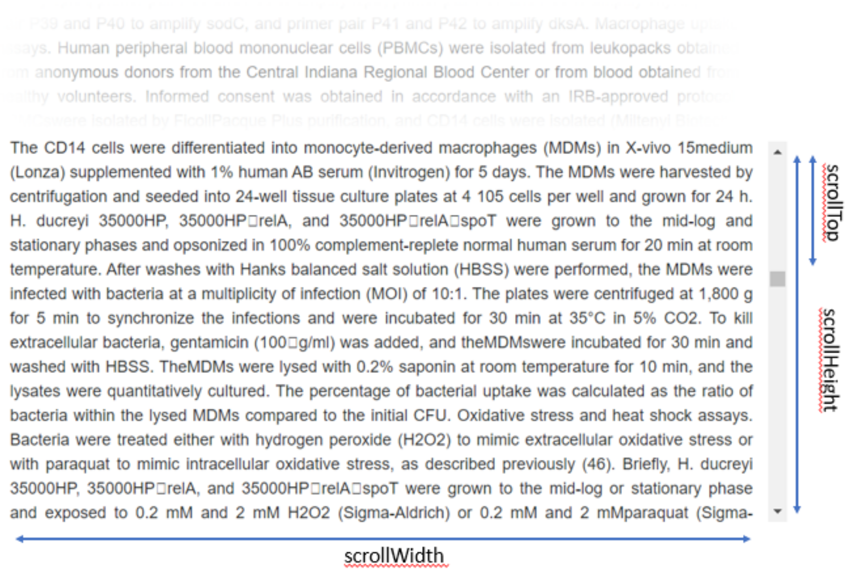
1. Define a reference variable in the template (e.g. <div #dataContainer).
2. Use angular *ViewChild* [9] *decorator*, as defined by *Angular's* property decorator that configures a *view query*. The change detector looks for the first element or the directive matching the selector in the *view* DOM to access the element in the DOM from the *component* (e.g. *@ViewChild('dataContainer') dataContainer: ElementRef;*).
3. Build a string with HTML tags and style that displays the *annotations* properly.
4. Assign the attribute *innerHTML* to the element referenced in the *template* with the string.

The first and second step on the enumeration are defined once, while the third and fourth steps are used whenever it is necessary to change the annotated text *annotations*. With *ngOnInit*, we get all the building blocks necessary to mark and style the text. However before tackling this process, it is necessary to address a major difference the *abstract* and *full text views* have.

When the function to annotate entities of a *IE process* in the abstract is called, it will locate and mark entities of that *IE process* through all of the abstract text extent. First, this was also valid for full text. However, we noticed the length on many of these documents caused an extended waiting period to annotate the text. The solution implemented is to only annotate entities within a certain range of the portion of text that is visible. A range is used for two reasons, the calculated offsets are estimations and to reduce the number of times the scroll experience is aggravated by the application annotation process to a chunk of text. The annotation of the text while scrolling has the following implementations:

- On the *view*: The div with the annotated text has an identifier the *controller* can use to reach the element, allow scrolling of the text and call a method defined in the *component* on scroll events.
- On the *controller*: With informations from the element defined in the *template*(*scrollWidth*, *scrollHeight*, *scrollTop*, *offsetHeight*) (Figure 22) and the length of the text, calculate the offsets where entities can be marked between:
    1. $topVolume * textLength / totalVolume = isInInd$ defines the estimation of the index in the text length of the first visible character: by multiplying *scrollWidth* with *scrollHeight*, we get the total volume of the element; By multiplying *scrollWidth* with *scrollHeight* we get the volume of text not shown on the top, the top volume. If we correlate the total volume to the text length we can estimate the text length of the top volume.

9 https://angular.io/api/core/ViewChild#usage-notes



P39 and P40 to amplify sodC, and primer pair P41 and P42 to amplify dksA. Macrophage up
ays. Human peripheral blood mononuclear cells (PBMCs) were isolated from leukopacks obtai
m anonymous donors from the Central Indiana Regional Blood Center or from blood obtained fr
lthy volunteers. Informed consent was obtained in accordance with an IRB-approved proto
MCswere isolated by FicollPacque Plus purification, and CD14 cells were isolated (Miltenyi Bi

The CD14 cells were differentiated into monocyte-derived macrophages (MDMs) in X-vivo 15medium (Lonza) supplemented with 1% human AB serum (Invitrogen) for 5 days. The MDMs were harvested by centrifugation and seeded into 24-well tissue culture plates at 4 105 cells per well and grown for 24 h. H. ducreyi 35000HP, 35000HP□relA, and 35000HP□relA□spoT were grown to the mid-log and stationary phases and opsonized in 100% complement-replete normal human serum for 20 min at room temperature. After washes with Hanks balanced salt solution (HBSS) were performed, the MDMs were infected with bacteria at a multiplicity of infection (MOI) of 10:1. The plates were centrifuged at 1,800 g for 5 min to synchronize the infections and were incubated for 30 min at 35°C in 5% CO2. To kill extracellular bacteria, gentamicin (100□g/ml) was added, and theMDMswere incubated for 30 min and washed with HBSS. TheMDMs were lysed with 0.2% saponin at room temperature for 10 min, and the lysates were quantitatively cultured. The percentage of bacterial uptake was calculated as the ratio of bacteria within the lysed MDMs compared to the initial CFU. Oxidative stress and heat shock assays. Bacteria were treated either with hydrogen peroxide (H2O2) to mimic extracellular oxidative stress or with paraquat to mimic intracellular oxidative stress, as described previously (46). Briefly, H. ducreyi 35000HP, 35000HP□relA, and 35000HP□relA□spoT were grown to the mid-log or stationary phase and exposed to 0.2 mM and 2 mM H2O2 (Sigma-Aldrich) or 0.2 mM and 2 mMparaquat (Sigma-

Figure 22.: Scroll properties

2. $visibleVolume * textLength / totalVolume = charsInVisible$ defines the estimation of number of characters visible in the text: the visible volume can be calculated by multiplying the *offsetHeight* with *scrollWidth*.
3. The start offset is estimated by subtracting a range value to the estimated *isInInd*. If the result is lower or equal to zero the start offset used is zero.
4. The end offset is estimated by adding *charsInVisible* to *isInInd* plus the range. If this value is superior to the text length the value used is the text length.

There are multiple operations to add *annotations* to the text, however the method responsible to add them, *buildAnnotatedText*, only annotates on text the *annotations* received as input. When on *full text view* methods that call *buildAnnotatedText*, with a subset of *annotations* between the offset values explained above.

The method *buildAnnotatedText* builds a string adding to the text *b* HTML elements representing the *annotation*. The *b* HTML tag is used at the start of the annotation offset, at the end offset is placed the respective closing tag. This HTML element is built with five variables each applied to a *attribute,attributes* provide additional information about the element:

1. *annotId*: references the annotation, applied to the id *attribute*.
2. *color*: color of the class, applied to the style *attribute*, parameter *color*.
3. *borderbottom*: thickness of the border, applied to the style *attribute*, parameter *border-bottom*.

4. *processColor*: color of the process, applied to the style *attribute*, parameter *border-bottom*.

5. *padding*: distance of the line to text, applied to the style *attribute*, parameter *padding-bottom*.

This element will color the text and create a line below it, allowing the representation of multiple *IE processes* by changing the padding for each line. However, for the padding not to interfere with the text below, only three levels of padding are allowed. This also limits an entity to be represented by the color of a single class, it will be of the resource in the last process selected, that isn't included in a previously annotated text. The *curator component* also has the feature to see informations about the annotated *terms* when hovering over them in a *tooltip*. The information displayed when applicable is: *class*; *term*; *id*; number of annotated synonyms in the text; *synonyms*; *external identifiers*. In the *view* the *divs* that can contain annotations call a function on *mouseover* and *mouseleave* events, that controll the display of the information described. The method receives data on the event in an input variable. The information is presented in a *div* created in the *controller*, with a *span* element for each *term*. The *span* elements have opacity and a black background to contrast with the annotated text. The *html* hovered can be accessed in the DOM information received as input. With that is known the *annotation* id and with the *process* color it simple to know the *process*. This allows to locate in the structures of the *component*, the data necessary to populate the *tooltip*. Since *tooltips* can have relevant information to copy, (e.g. an external database identifier), we decided the *tooltip* should stay visible even after the mouse leaves the *annotation* on the text. The implemented solution is: when the element is hovered the *tooltip* is generated. However it waits a small amount of time before it is added to the *view* and if the mouse pointer leaves before that time elapses the element is destroyed therefore not displayed. This allows the user to only access the *annotation* information when is intended preventing *tooltips* from blocking the text, when moving the cursor pointer and it hovers *annotations* in its path. When a *tooltip* is added to the *view*, it will remain until the user clicks anywhere outside of the *tooltip*. Only one section of *annotations* can be displayed at a time.

3.6.4 *Global search*

The *global search module* provides the users the ability to build a Boolean query using *resource elements* as building blocks, fetch the *documents* that match the query displaying them in a table. If the results are satisfying, a *corpus* can be created to run *IE processes* on that set of *documents*.

Global search query building

Building a query in global search is an interaction between the search for *resource elements* and the association of selected *resource elements* trough Boolean operators.

The search for *resource elements* uses the shared *search component*, with the respective configurations:

- *Search fields*: *Resource elements* can be searched by the *term* name, a synonym or by an identifier of an external database.
- *Search configuration*: The query can be executed as *keywords*, *whole sentence* or *suggestions*. It can also be specified if the query is *case-sensitive* or not.
- *Filters*: The *filter* is based on *resources* and *classes*.

The results of a search are added to a structure implemented using an external project, *angular-tree-component*. It allows to present the data in a tree structure, with features such as: keyboard navigation, expand/collapse/select nodes and drag & drop. These features are particularly useful for building the Boolean expression, the search results are also implemented with the *angular-tree-component* to enable the addition of *resource elements* from the search results *model* to the *query builder model* using the drag & drop feature. The list of results are shown in pages with a maximum of ten *resource elements* with the usual navigation options.

The *query builder* builds its *model* by allowing nodes from the *resource elements* results list and nodes from an available logic operators list to be dragged and dropped to the *query builder* space in the *view*. Nodes from the logical operators list (and, or), can have children nodes (when another node is dropped on top of it) and if it is double clicked it changes the logical operator (e.g. if the node has the and operator it changes to or and vice-versa). When the query is requested to be executed, the nodes in the *query builder model* are converted into a string written in the language explained at sub-subsection 3.4.2.

3.6.5 *Navigation*

Table 12 contains all the pages in the web application, with the respective routes. If an element in the *Navbar* column is different from *None*, it means it has a link in the web application *navbar*. The *tableParams* in route parameters coupled with their *components* interaction, allows to "save" and load the page with the table in a defined state (e.g. a table's results page advanced to page three, when opening a row element changing the route, after checking the content and clicking the back browser button, the table loads already in the results' page three).

Route	Route parameters	Description	Navbar
/	None	Home page	Home icon
/queries	tableParams	All queries the user has access	Queries
/queries/:id	id ->query id	View of a single query with the id on the route parameter	None
/copus	tableParams	All corpus the user has privileges	Corpus
/corpus/processes/:id	id ->corpus id	Corpus processes the user has privileges	None
/corpus/processes/:id/:pId	Id->corpus id; pId->process id; tableParams	Process details and associated documents	None
/documents	tableParams	All documents view	Documents
/publications/:type/:id	type->type of identifier e.g. id, doi etc. id->publication id of the given type	Document view	None
/processes	tableParams	All processes the user has access	Processes
/resources	tableParams	All resources the user has access	Resources
/resources/:resourceName/:classId	resourceName->resource name classId->resource id	Resource view	None
/curator	None	Document form to open curator view	Curator

/curatorview	processId->process id documentId->document id	processId->process id documentId->document id	None
/globalSearch	None	Boolean search of resource elements on all annotated documents	Global search
/globalSearch/queryResult/:query	query->Boolean query	Temporary page that makes the query request	None
/globalSearch/showResult	None	Shows the publications that comply with the query	None
/user	None	Edit personal information and administration	Profile icon and username
/login	None	Login	Login

Table 12.: Web application page routes

.

4 RESULTS

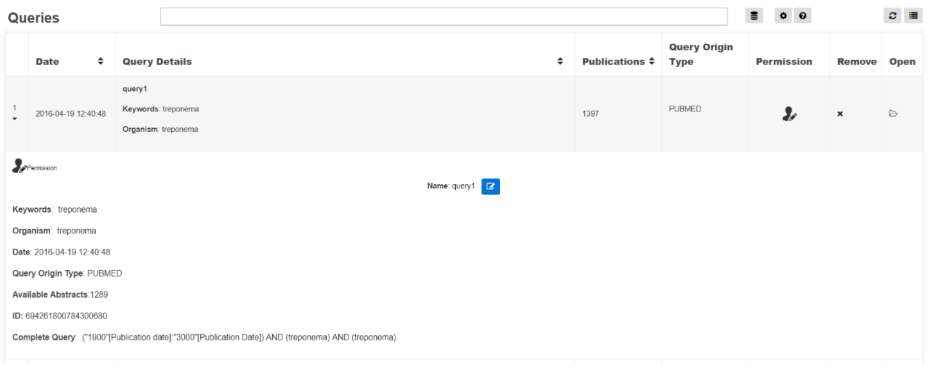
This chapter is dedicated to showing the web application running over an existing database. Each subsection will go through a domain of information or a major operation referencing the pages they are included in. It will go through the *@Note web* interface showing how the information is displayed and explaining the user interactions. To better illustrate user interactions, this chapter will have some examples of user inputs.

4.1 QUERIES

Queries are IR processes that fetch a set of *documents* of an external source given specific parameters. The *@Note* platform has several processes defined to query external databases, such as the *PubmedSearch*, that queries the *Pubmed* database. Such a *query* (query1) is the first element in the *queries* table shown in figure 23. Inspecting the first row of the table (query1), it was executed in the date: 2016-04-19, has the name query1, both the *keyword* and *organism* are *Treponema*, it has the identifier to its *query* type of *PUBMED* and it selected 1397 relevant *documents*. In the expanded row, it has more details on the *query* configuration, in the example, also showing a text encompassing the configurations related with Boolean operators at *Complete Query*. The *query* example doesn't have, however, additional configurations such as the inclusion of a date range. When the user has the respective permissions as a logged user, the name of the *query* can be edited.

The text search can be based on *Name*, *Keywords* and *Organism*, the three parameters used on this *query* (organism is an optional parameter). The available options are *Whole sentence*, *Keywords* and *Case-sensitive*.

Opening query1 changes to a page as presented by figure 24, it contains a header with the *query* name, a collapsible titled *Query informations*, which when it is expanded shows the same information of the associated expanded row on the *query* table, and a *publications* table, with the *publications* associated to the *query*. The unique property of the *publications* table on a *query* page is the relevance of the *publication* to the *query*, relevance has four levels of magnitude from lowest to greater they are: *None*, *Irrelevant*, *Related*, *Relevant*. Relevance levels are represented by the number of filled star icons, one filled icon means that the



Queries

	Date	Query Details	Publications	Query Origin Type	Permission	Remove	Open
1	2016-04-19 12:40:48	query1 Keywords: treponema Organism: treponema	1397	PUBMED		×	

Permission

Name: query1

Keywords: treponema

Organism: treponema

Date: 2016-04-19 12:40:48

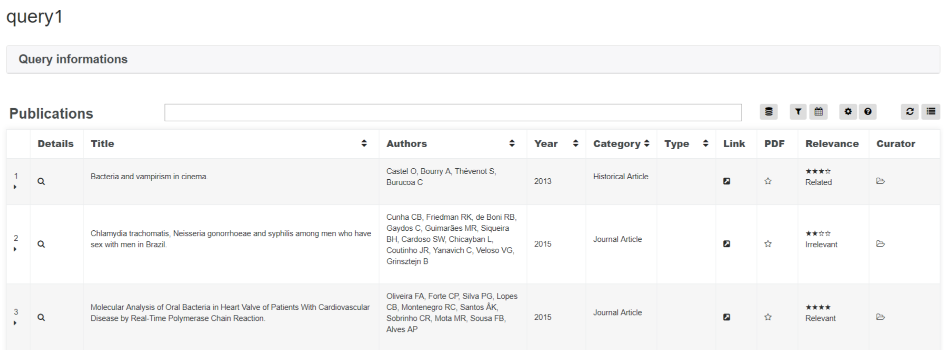
Query Origin Type: PUBMED

Available Abstracts:1289

ID: 694261800784300680

Complete Query: ("1900"[Publication date]:"3000"[Publication Date]) AND (treponema) AND (treponema)

Figure 23.: *@Note web Queries* interface



query1

Query informations

Publications

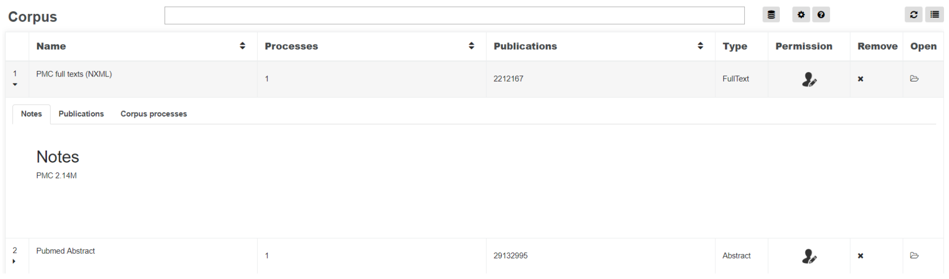
	Details	Title	Authors	Year	Category	Type	Link	PDF	Relevance	Curator
1		Bacteria and vampirism in cinema.	Castel O, Bourry A, Thévenot S, Burucoa C	2013	Historical Article			☆	★★★☆ Related	
2		Chlamydia trachomatis, Neisseria gonorrhoeae and syphilis among men who have sex with men in Brazil.	Cunha CB, Friedman RK, de Boni RB, Gaydos C, Guimarães MR, Siqueira BH, Cardoso SW, Chicayban L, Coutinho JR, Yanavich C, Veloso VG, Grinsztejn B	2015	Journal Article			☆	★★☆☆ Irrelevant	
3		Molecular Analysis of Oral Bacteria in Heart Valve of Patients With Cardiovascular Disease by Real-Time Polymerase Chain Reaction.	Oliveira FA, Forte CP, Silva PG, Lopes CB, Montenegro RC, Santos ÂK, Sobrinho CR, Mota MR, Sousa FB, Alves AP	2015	Journal Article			☆	★★★★ Relevant	

Figure 24.: *@Note web Query* interface

relevance is None the number than follows the sequence. The user can edit the relevance by selecting the number of filled stars.

4.2 CORPORA

Corpora are structures with documents associated over which information extraction processes can be applied. A *Corpus*, like many other entities, has a page with a tabular representation for its information. Figure 25 shows a table for a*corpus*, where the first row contains a *corpus* named PMC full texts (NXML), as the name implies the text type is *Full-Text*, contains 2212167 *documents* associated to it and one *IE process* run on those *documents*. By expanding the row, it provides access to a tabbed menu with three options: *Notes*, *Publications*, *Corpus processes*. In *Notes*, if the authenticated user has the respective permissions, she/he can edit the *corpus'* notes for future reference. In the example presented, the user only has *read* permission so only the text is shown. The *Publications* tab will contain a table with the documents contained by this *corpus*, similarly the *Corpus processes* tab contains a table, but instead of containing *documents* it has the run *IE processes*. By opening a *corpus*,



Corpus

	Name	Processes	Publications	Type	Permission	Remove	Open
1	PMC full texts (NXML)	1	2212167	FullText		×	
2	Pubmed Abstract	1	29132995	Abstract		×	

Notes | Publications | Corpus processes

Notes

PMC 2.14M

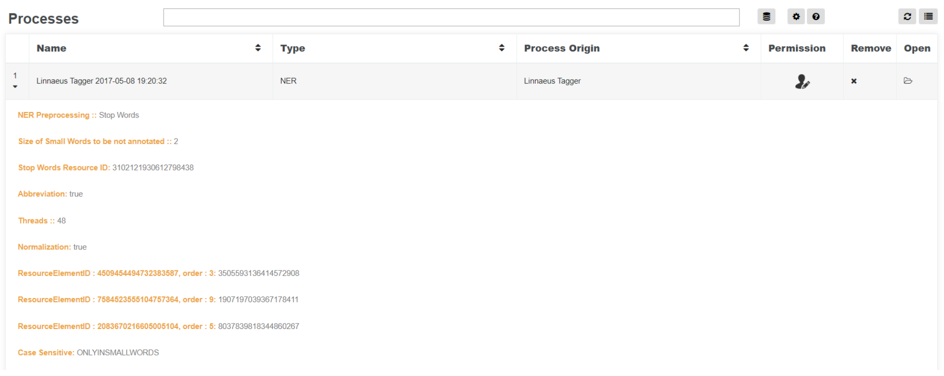
Figure 25.: *@Note web Corpus* interface

the application presents a page with the *processes* run on that *corpus* (the table for *processes* is explored in the next section). The text search can be based on *Name* and *Notes*, and its available options are *Whole sentence*, *Keywords* and *Case-sensitive*.

4.3 PROCESSES

Processes are BTM processes, that use resources to extract relevant information from natural text. They can be divided into two types: NER or RE. The *@Note* platform has several *processes* with different algorithms and configuration parameters. Figure 26 has a table for *processes*, focused on displaying the first row. In the row it displays that this *process* is called Linnaeus Tagger plus the date the process executed. It is an NER process and used the *Linnaeus Tagger process* configuration. In the expanded row space, it contains more information on how the *process* was configured. Continuing to explore the Linnaeus Tagger *process* in figure 26, it informs that the *annotations* accept abbreviations and words smaller than two characters are not annotated. It also shows resources used referencing its id, for example this *process* used a *Stop Words Resource* with the id: 3102121930612798438. The example shown does not display all the information, it misses some *resource elements* ids and a configurable parameter, *Disambiguation* that can be set to *ON* or *OFF*. The text search can be based on *Name*, *Type*, *Process Origin* and *Notes*, and its available options are *Whole sentence*, *Keywords* and *Case-sensitive*.

The page presented when opening a *process* is illustrated by figure 27, it has a collapsible titled *Process details*, which when expanded shows the same information of the associated expanded row on the *processes* table and a table with columns unique to *processes* named *Annotations*. The *Annotations* column contains a list of annotated entities with the respective number of times the entity is annotated in the *document*. In the example presented, all *annotations* are of proteins, the first *document* has 441, the second 7 and the third 488. The *documents* table is properly explained in the following section.



Processes

	Name	Type	Process Origin	Permission	Remove	Open
1	Linnaeus Tagger 2017-05-08 19:20:32	NER	Linnaeus Tagger		×	

NER Preprocessing :: Stop Words

Size of Small Words to be not annotated :: 2

Stop Words Resource ID: 3102121930612798438

Abbreviation: true

Threads :: 48

Normalization: true

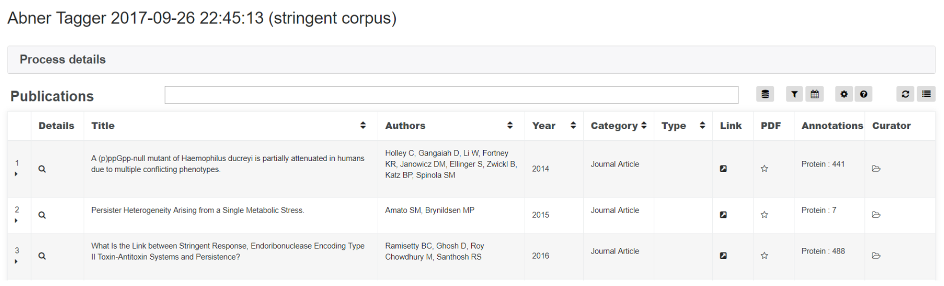
ResourceElementID : 4509454494732383587, order : 3: 3505593136414572908

ResourceElementID : 7584523555104757364, order : 9: 1907197039367178411

ResourceElementID : 2083670216605005104, order : 5: 8037839818344860267

Case Sensitive: ONLYINSMALLWORDS

Figure 26.: *@Note web Processes* interface



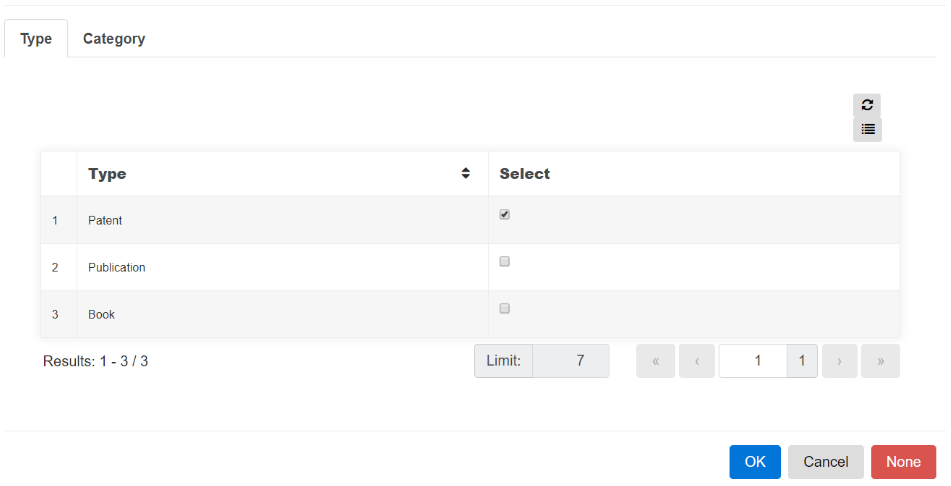
Abner Tagger 2017-09-26 22:45:13 (stringent corpus)

Process details

Publications

	Details	Title	Authors	Year	Category	Type	Link	PDF	Annotations	Curator
1		A (p)ppGpp-null mutant of Haemophilus ducreyi is partially attenuated in humans due to multiple conflicting phenotypes.	Holley C, Gangaiah D, Li W, Fortney KR, Janowicz DM, Ellinger S, Zwickl B, Katz BP, Spinola SM	2014	Journal Article			☆	Protein : 441	
2		Persister Heterogeneity Arising from a Single Metabolic Stress.	Amato SM, Brynildsen MP	2015	Journal Article			☆	Protein : 7	
3		What Is the Link between Stringent Response, Endoribonuclease Encoding Type II Toxin-Antitoxin Systems and Persistence?	Ramisetty BC, Ghosh D, Roy Chowdhury M, Santhosh RS	2016	Journal Article			☆	Protein : 488	

Figure 27.: *@Note web Process* interface



	Type	Select
1	Patent	✓
2	Publication	☐
3	Book	☐

Results: 1 - 3 / 3   Limit: 7   « ‹ 1 1 › »

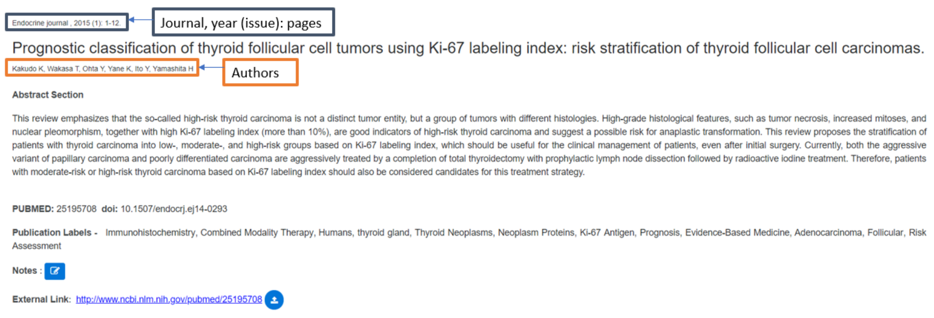
Figure 28.: *@Note web Document* filter *modal* interface

4.4 DOCUMENTS

Documents are the base from where BTM processes extract knowledge. As the number for relevant documents increases rapidly over the years, a powerful search engine is valuable to quickly check the database on a study subject. The implemented search engine does not consider synonyms of words or tries to correctly guess abbreviations, however it can give a rough approximation and it is easy to iterate multiple search operations. *Publications* tables text search can be based on *Title*, *Authors*, *Journal*, *Abstract Section*, *Category*, *Type*, *Notes*, *Full Text Content*. The available options are *Filters*, *Year*, *Whole sentence*, *Keywords*, *Case-sensitive* and *Expression*. The available filters are *Type* (currently documents can be divided into three types: *Patent*, *Publication*, *Book*) and *Category* (examples of categories are: Journal Article, English Abstract and Case Reports). Figure 28 shows the filter *modal* interface with the *Patent Type* selected.

By expanding a row or clicking on the *Open* button, document metadata is available. In the example presented in the figure 29, we have a *publication* of the *Endocrine* journal released in the year 2015, issue 1, authored by Kakudo et al. From the title and abstract, we get to know this is a review about patient classification related to thyroid carcinoma and ki-67 labeling index. The labels can help to quickly understand the content of the document. The page also contains external identifiers of the *document*, in this case we have the identifier for the pubmed database and it's DOI. Also, a there is a place to view and edit notes when applicable and an external link (that can be updated by clicking on the icon on its side).

In each *publications* table, it is possible to create a corpus after a search by pressing the button *Create corpus* that will appear as presented in figure 30. Then the *corpus* name and notes can be filled and, finally, submitted to start the *corpus* creation process. Processes



Endocrine journal , 2015 (1): 1-12.

Prognostic classification of thyroid follicular cell tumors using Ki-67 labeling index: risk stratification of thyroid follicular cell carcinomas.

Kakudo K, Wakasa T, Ohta Y, Yane K, Ito Y, Yamashita H

Abstract Section

This review emphasizes that the so-called high-risk thyroid carcinoma is not a distinct tumor entity, but a group of tumors with different histologies. High-grade histological features, such as tumor necrosis, increased mitoses, and nuclear pleomorphism, together with high Ki-67 labeling index (more than 10%), are good indicators of high-risk thyroid carcinoma and suggest a possible risk for anaplastic transformation. This review proposes the stratification of patients with thyroid carcinoma into low-, moderate-, and high-risk groups based on Ki-67 labeling index, which should be useful for the clinical management of patients, even after initial surgery. Currently, both the aggressive variant of papillary carcinoma and poorly differentiated carcinoma are aggressively treated by a completion of total thyroidectomy with prophylactic lymph node dissection followed by radioactive iodine treatment. Therefore, patients with moderate-risk or high-risk thyroid carcinoma based on Ki-67 labeling index should also be considered candidates for this treatment strategy.

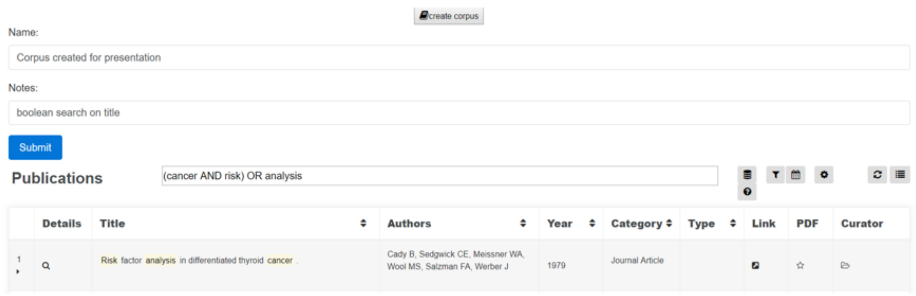
PUBMED: 25195708 **doi:** 10.1507/endocrj.ej14-0293

Publication Labels - Immunohistochemistry, Combined Modality Therapy, Humans, thyroid gland, Thyroid Neoplasms, Neoplasm Proteins, Ki-67 Antigen, Prognosis, Evidence-Based Medicine, Adenocarcinoma, Follicular, Risk Assessment

Notes :

External Link: http://www.ncbi.nlm.nih.gov/pubmed/25195708

Figure 29.: *@Note web Document* interface



Name: Corpus created for presentation

Notes: boolean search on title

Publications (cancer AND risk) OR analysis

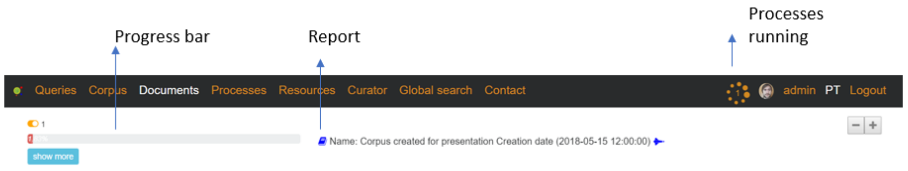
	Details	Title	Authors	Year	Category	Type	Link	PDF	Curator
1		Risk factor analysis in differentiated thyroid cancer .	Cady B, Sedgwick CE, Meissner WA, Wool MS, Salzman FA, Werber J	1979	Journal Article			☆	

Figure 30.: *@Note web Corpus* creation by *document* expression search interface

progress are shown on the top of the page. Figure 31 illustrates how this information is displayed. If a related process to the page is running, the progress bar fills accordingly to the current percentage of progress. The report also changes with the state of the process, which when finished provides additional information.

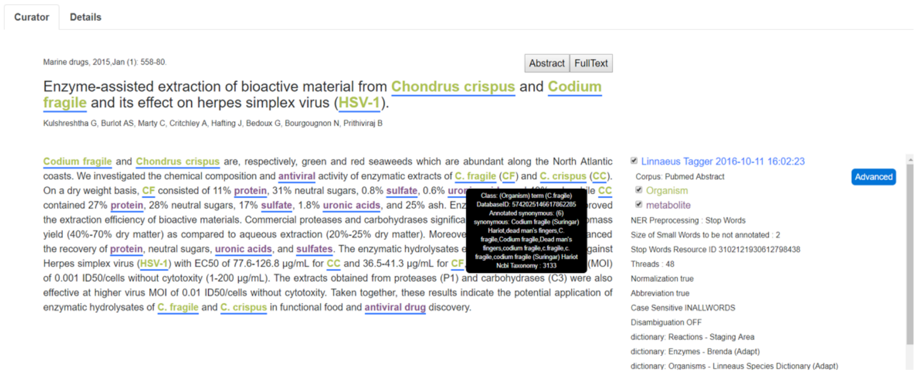
4.5 CURATOR

The *curator* page, in its current state, does not allow for any editing of information. However, it has the information on annotated *terms*, *resources* and *processes* on the *document* visually integrated with the text. By clicking on a process, in the *processes'* list at the right, under it will have the associated *corpus* and *classes* involved, each class with its defined color. Both



Name: Corpus created for presentation Creation date (2018-05-15 12:00:00)

Figure 31.: *@Note web* running process on top of page interface



Marine drugs, 2015,Jan (1): 558-80.

Enzyme-assisted extraction of bioactive material from Chondrus crispus and Codium fragile and its effect on herpes simplex virus (HSV-1).

Kulshreshtha G, Burlot AS, Marty C, Critchley A, Hafting J, Bedoux G, Bourgougnon N, Prithiviraj B

Codium fragile and Chondrus crispus are, respectively, green and red seaweeds which are abundant along the North Atlantic coasts. We investigated the chemical composition and antiviral activity of enzymatic extracts of C. fragile (CF) and C. crispus (CC). On a dry weight basis, CF consisted of 11% protein, 31% neutral sugars, 0.8% sulfate, 0.6% uro le CC contained 27% protein, 28% neutral sugars, 17% sulfate, 1.8% uronic acids, and 25% ash. Enz roved the extraction efficiency of bioactive materials. Commercial proteases and carbohydrases significa mass yield (40%-70% dry matter) as compared to aqueous extraction (20%-25% dry matter). Moreove anced the recovery of protein, neutral sugars, uronic acids, and sulfates. The enzymatic hydrolysates e gainst Herpes simplex virus (HSV-1) with EC50 of 77.6-126.8 μg/mL for CC and 36.5-41.3 μg/mL for CF (MOI) of 0.001 ID50/cells without cytotoxity (1-200 μg/mL). The extracts obtained from proteases (P1) and carbohydrases (C3) were also effective at higher virus MOI of 0.01 ID50/cells without cytotoxity. Taken together, these results indicate the potential application of enzymatic hydrolysates of C. fragile and C. crispus in functional food and antiviral drug discovery.

Class: (Organism) term (C.fragile)
DatabaseID: 5742025146617862285
Annotated synonymous: (6)
synonymous: Codium fragile (Suringar) Hariot,dead man's fingers,C. fragile,Codium fragile,Dead man's fingers,codium fragile,c.fragile,c. fragile,codium fragile (Suringar) Hariot
Ncbi Taxonomy : 3133

☑ Linnaeus Tagger 2016-10-11 16:02:23
Corpus: Pubmed Abstract
☑ Organism
☑ metabolite
NER Preprocessing : Stop Words
Size of Small Words to be not annotated : 2
Stop Words Resource ID 3102121930612798438
Threads : 48
Normalization true
Abbreviation true
Case Sensitive INALLWORDS
Disambiguation OFF
dictionary: Reactions - Staging Area
dictionary: Enzymes - Brenda (Adapt)
dictionary: Organisms - Linneaus Species Dictionary (Adapt)

Figure 32.: *@Note* curator abstract interface

the process and colors can be selected to interact with the text coloring and underlining the respective entities.

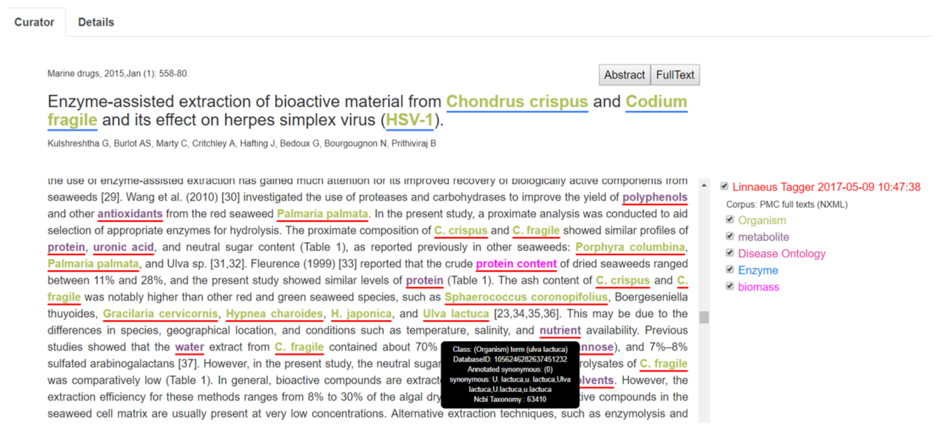
Figure 32 shows the *curator* page for a *document* on *abstract view*. It has the *Linnaeus Tagger* process selected, to show the annotated organisms. Similar to the *process* details on *processes* tables, if the advanced button is clicked, it shows information on the configurations and *resources* used (e.g. this *process* uses a Chebi Ontology to find metabolites and a Linnaeus Species Dictionary for organisms).

In the text since only one *process* is selected all annotated entities have a blue underline matching the *process'* color. The text color of the *annotations* on text facilitates in identifying entities of interest, its also evident which of these are metabolites or organisms. In the example the organism C.fragile is hovered, displaying a tooltip with black background and white text. The information on the term and it's synonyms can be particularly useful, when the hovered entity is an abbreviation or a name the user doesn't recognize. With the Ncbi taxonomy id, the user can go to the Ncbi website and learn more on this organism.

Figure 33 has the same *document*, in the *full text view*, in the *processes* list it shows that the *process* is also named Linnaeus Tagger (different date), a referencing the same algorithm. However it also shows that this *process* is applied to a different *corpus*. This the *FullText* type *corpus*, used in the example on section 4.2. In the *full text view* the *process* managed to annotate diseases, enzymes and biomass, *class* types the *process* in the *process* in the *abstract view* didn't.

4.6 Resources

Resources are fundamental pieces to BTM processes, and they usually need to be updated to cope with new discoveries or changes. The web application gives access to five types of *resources*: Dictionary, Lookup table, Rule Set, Ontology and Lexical words. The *resources* of


Curator | Details

Marine drugs, 2015,Jan (1): 558-80.

Abstract | FullText

Enzyme-assisted extraction of bioactive material from Chondrus crispus and Codium fragile and its effect on herpes simplex virus (HSV-1).

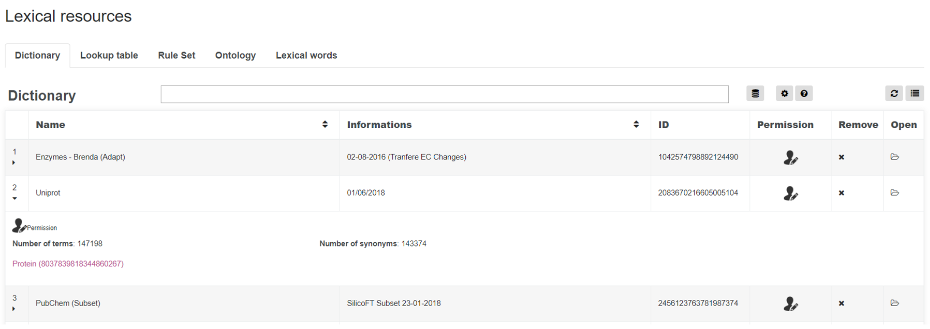
Kulshreshtha G, Burlot AS, Marty C, Critchley A, Hafting J, Bedoux G, Bourgougnon N, Prithiviraj B

the use of enzyme-assisted extraction has gained much attention for its improved recovery of biologically active components from seaweeds [29]. Wang et al. (2010) [30] investigated the use of proteases and carbohydrases to improve the yield of polyphenols and other antioxidants from the red seaweed Palmaria palmata. In the present study, a proximate analysis was conducted to aid selection of appropriate enzymes for hydrolysis. The proximate composition of C. crispus and C. fragile showed similar profiles of protein, uronic acid, and neutral sugar content (Table 1), as reported previously in other seaweeds: Porphyra columbina, Palmaria palmata, and Ulva sp. [31,32]. Fleurence (1999) [33] reported that the crude protein content of dried seaweeds ranged between 11% and 28%, and the present study showed similar levels of protein (Table 1). The ash content of C. crispus and C. fragile was notably higher than other red and green seaweed species, such as Sphaerococcus coronopifolius, Boergeseniella thuyoides, Gracilaria cervicornis, Hypnea charoides, H. japonica, and Ulva lactuca [23,34,35,36]. This may be due to the differences in species, geographical location, and conditions such as temperature, salinity, and nutrient availability. Previous studies showed that the water extract from C. fragile contained about 70% nnose), and 7%–8% sulfated arabinogalactans [37]. However, in the present study, the neutral sugar rolysates of C. fragile was comparatively low (Table 1). In general, bioactive compounds are extract lvents. However, the extraction efficiency for these methods ranges from 8% to 30% of the algal dry ive compounds in the seaweed cell matrix are usually present at very low concentrations. Alternative extraction techniques, such as enzymolysis and

Class: (Organism) term (ulva lactuca)
DatabaseID: 1056246282637451232
Annotated synonymous: (0)
synonymous: U. lactuca,u. lactuca,Ulva lactuca,U.lactuca,u.lactuca
Ncbi Taxonomy : 63410

☑ Linnaeus Tagger 2017-05-09 10:47:38
Corpus: PMC full texts (NXML)
☑ Organism
☑ metabolite
☑ Disease Ontology
☑ Enzyme
☑ biomass

Figure 33.: *@Note web* curator full text interface


Lexical resources

Dictionary | Lookup table | Rule Set | Ontology | Lexical words

Dictionary

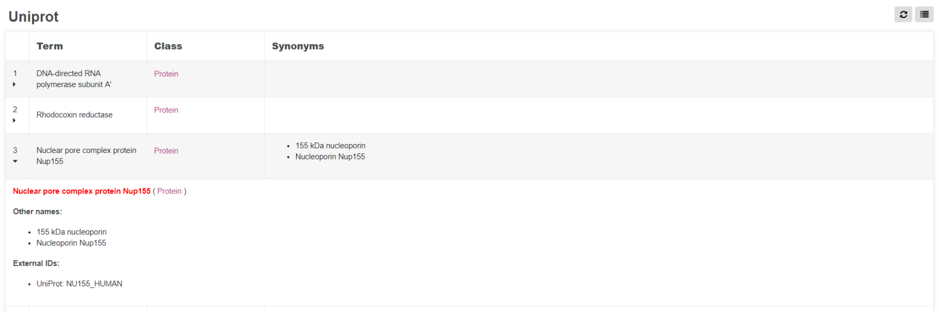
	Name	Informations	ID	Permission	Remove	Open
1	Enzymes - Brenda (Adapt)	02-08-2016 (Tranfere EC Changes)	1042574798892124490		✕	
2	Uniprot	01/06/2018	2083670216605005104		✕	
3	PubChem (Subset)	SilicoFT Subset 23-01-2018	2456123763781987374		✕	

Permission
Number of terms: 147198 **Number of synonyms**: 143374
Protein (8037839818344860267)

Figure 34.: @Note web Resources interface

each type are divided in a tab menu, containing a search data table with the resources of the corresponding type. The data table columns are: *Name*, *Information*, *ID*, *Permission* - button with the permissions configurations panel for *resources*, *Remove* - operation to remove the *resource*, *Open* - opens a *view* associated to the selected *resource*. The expanded row contains information on the number of terms and synonyms in the *resource*, and a list of all *classes* with their id. The text search can be based on Name and Information; its available options are *Whole sentence*, *Keywords* and *Case-sensitive*.

In figure 34, the tab menu is set on the *resource* type Dictionary, as such the search data table contains *resources* of that type. In the Uniprot dictionary, the row is expanded showing that it contains 147198 terms and 143374 synonyms, all proteins. Opening a resource changes to a page illustrated by figure 35, that contains a table for a resource, its columns are: *Term*, *Class*, *Synonyms*. The expandable row gives information on the synonyms and external Ids.



Uniprot

	Term	Class	Synonyms
1	DNA-directed RNA polymerase subunit A'	Protein	
2	Rhodocoxin reductase	Protein	
3	Nuclear pore complex protein Nup155	Protein	• 155 kDa nucleoporin • Nucleoporin Nup155

Nuclear pore complex protein Nup155 (Protein)

Other names:

- 155 kDa nucleoporin
- Nucleoporin Nup155

External IDs:

- UniProt: NU155_HUMAN

Figure 35.: *@Note web Resource* interface

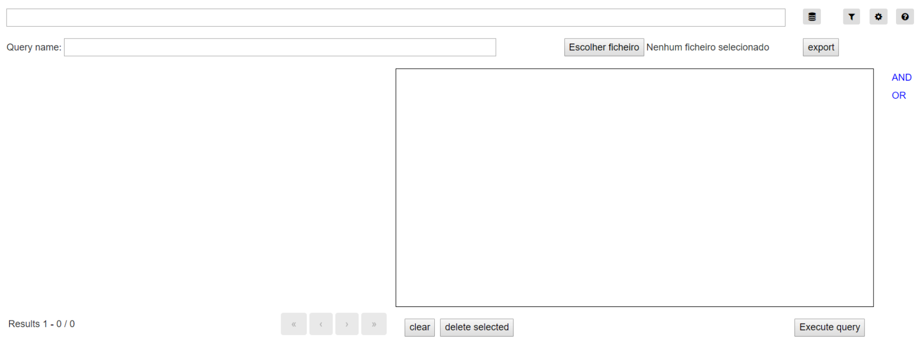


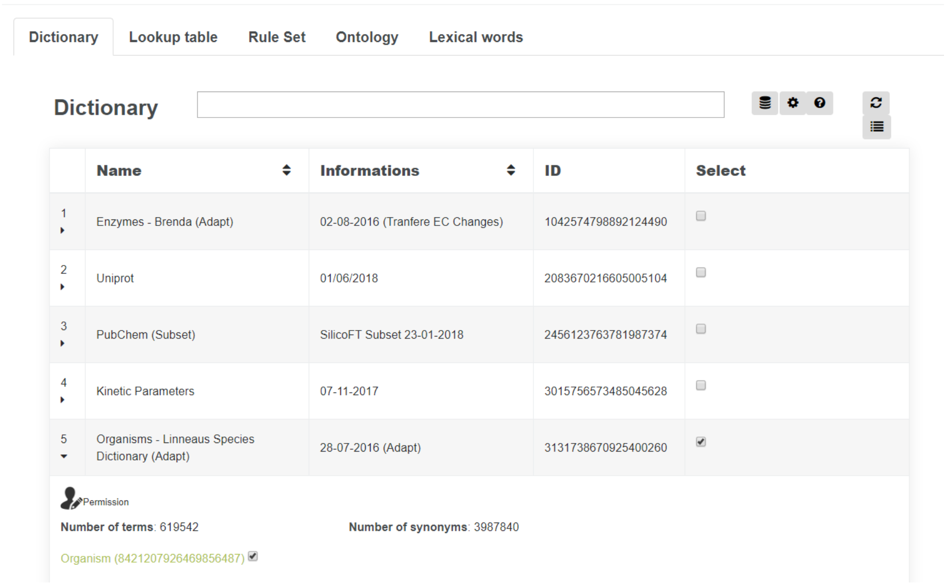
Figure 36.: *@Note web global search* initial state interface

4.7 GLOBAL SEARCH

The *globalSearch* page is dedicated to form a Boolean expression of *resource elements* contained in *resources*. The expression can be evaluated returning the publications with annotated *terms* that match the query.

As presented in figure 36 at the top, we have a search field on resource elements, where the fields available are: *Name*, *Synonyms* and *External Ids*, with the available options *Filters*, *Whole sentence*, *Keywords*, *Suggestions* and *Case-sensitive*. The filters model contains a tab menu with the available resource types, each containing a search data table of resource, with the option to select a resource for the filter. If the resource has multiple classes, by expanding its row classes, each class can be selected individually for the filter. In the example presented in figure 37, the results will be filtered by organism.

The search results will show on the left empty space as a list, each row containing the term followed by the class with parenthesis finished by a maximum of two synonyms (if the term has any). Rows can be expanded showing all other names and external identifiers. In figure 38, the *Ulva lactuca* organism (an algae) in the *resource elements* list, was added to the *query builder* by pressing and holding, and dragged to the dedicated space for the query



Dictionary

	Name	Informations	ID	Select
1	Enzymes - Brenda (Adapt)	02-08-2016 (Tranfere EC Changes)	1042574798892124490	☐
2	Uniprot	01/06/2018	2083670216605005104	☐
3	PubChem (Subset)	SilicoFT Subset 23-01-2018	2456123763781987374	☐
4	Kinetic Parameters	07-11-2017	3015756573485045628	☐
5	Organisms - Linneaus Species Dictionary (Adapt)	28-07-2016 (Adapt)	3131738670925400260	✓

Permission

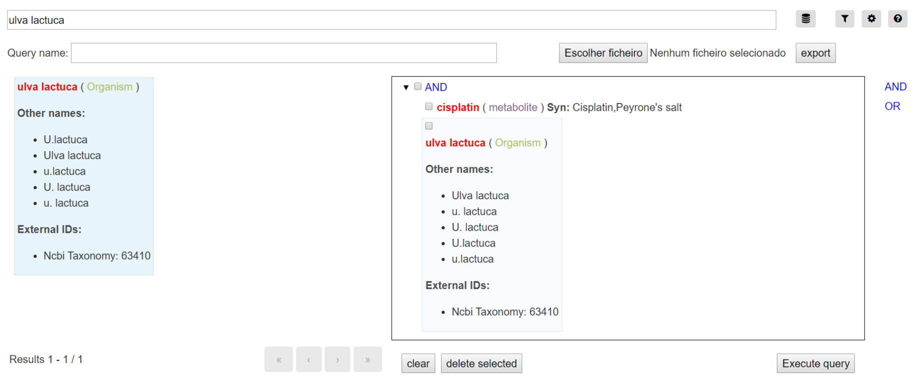
Number of terms: 619542 **Number of synonyms**: 3987840

Organism (8421207926469856487) ✓

Figure 37.: *@Note web global search* Filters interface

building at its right. At the far right are the available Boolean operators that can also be dragged and dropped into the query building space. The building space works as a tree with levels, the empty space is the first level and works as an OR operator where terms and operators can be added. Terms already present in a level will not be possible to add (the cursor icon will change to ”forbidden operation”). When operators have other elements attached to them, they will have the option to be collapsed or extended. All elements have the option to be selected, while term elements have the same option to be expanded as they were in the list of resource elements.

As an example, we show how to find documents containing *Ulva lactuca* and *cisplatin*, used in cancer treatment. The query is executed by pressing the button *Execute query*. A new tab opens in the browser with the page that contains a table with the resulting *documents* of the query. These results are shown in figure 39. The first document is a study that evaluates the toxicity and bioaccumulation of *cisplatin* in the marine environment using *Ulva lactuca*. In the second, we have a study related to cancer and algae. Both are relevant to the domain of the search, however they can be irrelevant to more specific subjects of study. This query had 11 results; it can be refined to either narrow the results even further or broaden them (e.g. add more cancer related entities, or other algae). It is possible to create a corpus with these documents by pressing the *Create corpus* button.



ulva lactuca

Query name:

Escolher ficheiro Nenhum ficheiro selecionado export

ulva lactuca (Organism)

Other names:

- U.lactuca
- Ulva lactuca
- u.lactuca
- U. lactuca
- u. lactuca

External IDs:

- Ncbi Taxonomy: 63410

▾ AND

- **cisplatin** (metabolite) **Syn:** Cisplatin,Peyrone's salt
- **ulva lactuca** (Organism)

  Other names:

  - Ulva lactuca
  - u. lactuca
  - U. lactuca
  - U.lactuca
  - u.lactuca

  External IDs:

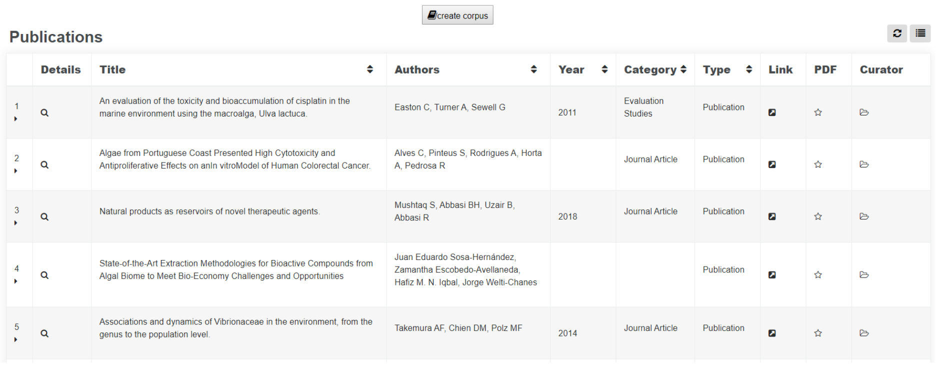
  - Ncbi Taxonomy: 63410

AND
OR

Results 1 - 1 / 1

clear delete selected Execute query

Figure 38.: *@Note web global search* interface building a query

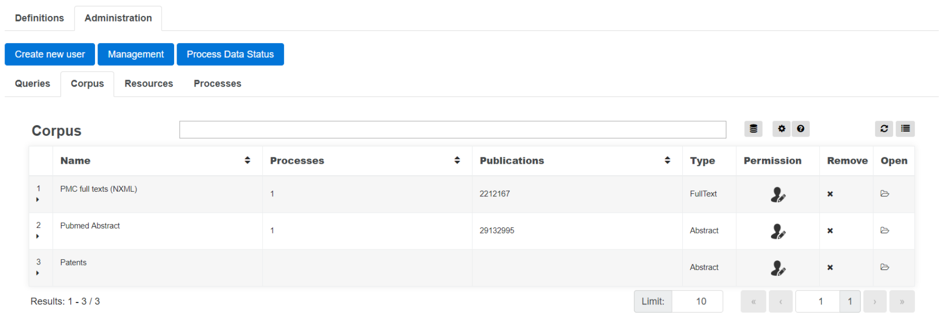


create corpus

Publications

	Details	Title	Authors	Year	Category	Type	Link	PDF	Curator
1		An evaluation of the toxicity and bioaccumulation of cisplatin in the marine environment using the macroalga, Ulva lactuca.	Easton C, Turner A, Sewell G	2011	Evaluation Studies	Publication		☆	
2		Algae from Portuguese Coast Presented High Cytotoxicity and Antiproliferative Effects on anIn vitroModel of Human Colorectal Cancer.	Alves C, Pinteus S, Rodrigues A, Horta A, Pedrosa R		Journal Article	Publication		☆	
3		Natural products as reservoirs of novel therapeutic agents.	Mushtaq S, Abbasi BH, Uzair B, Abbasi R	2018	Journal Article	Publication		☆	
4		State-of-the-Art Extraction Methodologies for Bioactive Compounds from Algal Biome to Meet Bio-Economy Challenges and Opportunities	Juan Eduardo Sosa-Hernández, Zamantha Escobedo-Avellaneda, Hafiz M. N. Iqbal, Jorge Welti-Chanes			Publication		☆	
5		Associations and dynamics of Vibrionaceae in the environment, from the genus to the population level.	Takemura AF, Chien DM, Polz MF	2014	Journal Article	Publication		☆	

Figure 39.: *@Note web global search* results interface

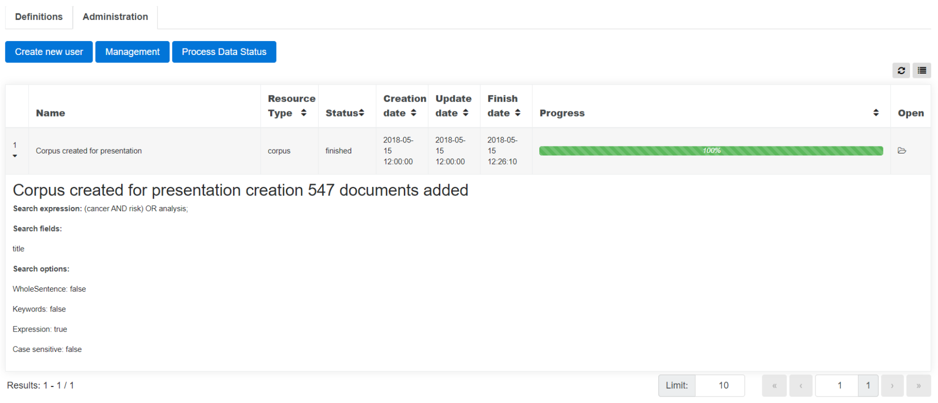


Corpus

	Name	Processes	Publications	Type	Permission	Remove	Open
1	PMC full texts (NXML)	1	2212167	FullText		×	
2	Pubmed Abstract	1	29132995	Abstract		×	
3	Patents			Abstract		×	

Results: 1 - 3 / 3 Limit: 10

Figure 40.: *@Note web* User administration management interface



	Name	Resource Type	Status	Creation date	Update date	Finish date	Progress	Open
1	Corpus created for presentation	corpus	finished	2018-05-15 12:00:00	2018-05-15 12:00:00	2018-05-15 12:26:10	100%	

Corpus created for presentation creation 547 documents added

Search expression: (cancer AND risk) OR analysis;

Search fields:

title

Search options:

WholeSentence: false

Keywords: false

Expression: true

Case sensitive: false

Results: 1 - 1 / 1 Limit: 10

Figure 41.: *@Note web* User administration process data status interface

4.8 USER

On the *@Note web* platform, users can edit their personal information by accessing the user page like updating their address, photo or change the password. The page also contains the administration menu where the admin can create new users, and all users can access *queries*, *corpus*, *resources* and *processes* available to him and change permissions when applicable (figure 40). The last option is to check the running and finished processes on the *process data status* table, as shown in figure 41.

5 CONCLUSIONS AND FUTURE WORK

In this chapter, we will cover the main conclusions of this work, the limitations of the application developed and the prospects for future work.

5.1 SUMMARY OF THE WORK

This work is based on the BTM *@Note* project, which provides the most important tasks of IR and IE. The IR methods select relevant documents, that can be grouped in a corpus. Documents in a corpus can have knowledge extracted by IE processes.

The supporting *@Note Java libs* were developed using several frameworks, such as *Hibernate Search*, *Hibernate ORM*, and *Spring*. To develop the *@Note web* application, we started by studying the relevant technologies (including *Angular* for *front-end* web development) and the *Java libs* implementation for the back-end.

The *@Note web* application developed focuses on the visualization and navigation of the main entities and results of BTM tasks. It also implements two IR methods that select documents by building a Boolean query. It uses recognizable icons and buttons to communicate their function, coupled with organized information in tables with varied manipulation of data options. *Global search* provides an intuitive searching and building system to use *resource elements* to search for relevant documents. The *curator* uses colors and tooltips to communicate relevant information on the *annotations*. It also addresses some management issues, such as allowing to change permissions and check the progress of running processes.

5.2 PROSPECTS FOR FUTURE WORK

Since *@Note web* focuses on the visualization aspect of the *@Note* framework, it lacks the implementation of many operations. Some of the operations that can be added are: the addition of new *resources* and their *resource elements*; *queries* to an external database; configuration of IE processes; a pipeline covering IR, NER and RE; manual curation of *documents*, and validation of *annotations* generated by *IE processes*.

On the search aspect of the application, it lacks *fields* to search by suggestions, the only implementation is in *global search*. This should be added to *fields* containing a few text. We could also expand on the configuration of the search, (e.g. provide the user with the ability to sort the importance of *keywords* or *phrases* that have an impact on the sorting of results). Another possibility would be to add a list of words on the domain of the search that boost results based on the quantity of words they contain.

On the visualization of annotated entities in *documents*, we could add visual cues for relations.

On *data process status*, possible improvements are: to add to running process methods a record of their progress and report on the error/success. For this, they need to interact with the *IDataProcessStatusAccess* interface (e.g. corpus creation explained at subsection 3.5.2).

On *global search*, we could consider the addition of more Boolean operators (or), a proper method to save and load Boolean queries, and a paginated method for *global search* query results.

BIBLIOGRAPHY

Hibernate orm. https://hibernate.org/orm/, a. Accessed: 2019-10-19.

Hibernate search. https://hibernate.org/search/, b. Accessed: 2019-10-19.

npm about. https://docs.npmjs.com/about-npm/. Accessed: 2019-10-22.

Spring cache tutorial. https://www.baeldung.com/spring-cache-tutorial, a. Accessed: 2019-10-21.

Spring cors support. https://spring.io/blog/2015/06/08/cors-support-in-spring-framework, b. Accessed: 2019-10-22.

w3 html/css. https://www.w3.org/standards/webdesign/htmlcss. Accessed: 2019-10-22.

Olivier Bodenreider. Lexical, Terminological and Ontological Resources for Biological Text Mining. *Text Mining for Biology and Biomedicine*, (Icd):43–66, 2006. doi: 10.1.1.76.1953.

Allen C Browne, Guy Divita, Alan R Aronson, and Alexa T Mccray. UMLS Language and Vocabulary Tools AMIA 2003 Open Source Expo. *AMIA 2003 Symposium Proceesings*, page 798, 2003.

David Campos, Jóni Lourenço, Tiago Nunes, Rui Vitorino, Pedro Sérgio Matos Domingues, and José Luís Oliveira. Egas - Collaborative Biomedical Annotation as a Service. *Fourth BioCreative Challenge Evaluation Workshop*, pages 254–259, 2013.

K. Bretonnel Cohen and Lawrence Hunter. Getting started in text mining. *PLoS Computational Biology*, 4(1):0001–0003, 2008. ISSN 1553734X. doi: 10.1371/journal.pcbi.0040020.

Maria Cristina ENACHE. Web Application Frameworks. *Economics and Applied Informatics*, (3):82–86, 2015.

K Fukuda, T Tsunoda, A Tamura, and T Takagi. Information Extraction: {I}dentifying Protein Names from Biological Papers. *Proceedings of the 3rd Pacific Symposium on Biocomputing*, pages 707–718, 1998. ISSN 1793-5091.

GeneOntologyConsortium. The Gene Ontology (GO) database and informatics resource. *Nucleic Acids Research*, 32(Database issue):258D–261, 2004. ISSN 1362-4962. doi: 10.1093/nar/gkh036.

James Gosling and Henry McGilton. The Java Language Environment. *Language*, (May):4, 1995.

J. D. Kim, T. Ohta, Y. Tateisi, and J. Tsujii. GENIA corpus - A semantically annotated corpus for bio-textmining. *Bioinformatics*, 19(SUPPL. 1):180–182, 2003. ISSN 13674803. doi: 10.1093/bioinformatics/btg1023.

Glenn E Krasner and Stephen T Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of object oriented programming*, 1(3): 26–49, 1988. ISSN 0896-8438. doi: 10.1.1.47.366.

Peder Olesen Larsen and Markus von Ins. The rate of growth in scientific publication and the decline in coverage provided by science citation index. *Scientometrics*, 84(3):575–603, 2010. ISSN 01389130. doi: 10.1007/s11192-010-0202-z.

Anália Lourenço, Rafael Carreira, Sónia Carneiro, Paulo Maia, Daniel Glez-Peña, Florentino Fdez-Riverola, Eugénio C. Ferreira, Isabel Rocha, and Miguel Rocha. @Note: A workbench for Biomedical Text Mining. *Journal of Biomedical Informatics*, 42(4):710–720, 2009. ISSN 15320464. doi: 10.1016/j.jbi.2009.04.002.

George a. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38 (11):39–41, 1995. ISSN 00010782. doi: 10.1145/219717.219748.

Chikashi Nobata, Philip Cotter, Naoaki Okazaki, Brian Rea, Yutaka Sasaki, Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. Kleio: a knowledge-enriched information retrieval system for biology. *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 787–788, 2008. doi: 10.1145/1390334.1390504.

Elizabeth O'Neil. Object/Relational mapping 2008: Hibernate and the entity data model (EDM). *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1351–1356, 2008. ISSN 07308078. doi: 10.1145/1376616.1376773.

Carolina Perez-Iratxeta, Peer Bork, and Miguel A Andrade. XplorMed: a tool for exploring MEDLINE abstracts. *Trends in Biochemical Sciences*, 26(9):573–575, sep 2001. ISSN 09680004. doi: 10.1016/S0968-0004(01)01926-0.

Pivotal. Spring Framework Reference Documentation. 2017.

Piotr Przybyła, Matthew Shardlow, Sophie Aubin, Robert Bossy, Richard Eckart de Castilho, Stelios Piperidis, John McNaught, and Sophia Ananiadou. Text mining resources for the life sciences. *Database : the journal of biological databases and curation*, 2016:1–15, 2016. ISSN 1758-0463. doi: 10.1093/database/baw145.

Rafal Rak, Andrew Rowley, William Black, and Sophia Ananiadou. Argo: An integrative, interactive, text mining-based workbench supporting curation. *Database*, 2012:1–7, 2012. ISSN 17580463. doi: 10.1093/database/bas010.

Raul Rodriguez-Esteban. Biomedical text mining and its applications. *PLoS Computational Biology*, 5(12):1–5, 2009. ISSN 1553734X. doi: 10.1371/journal.pcbi.1000597.

Hagit Shatkay and Mark Craven. *Mining the Biomedical Literature Computational Molecular Biology*. 2012. ISBN 9780262017695.

Lauren Wood and D O M Level. Document Object Model (DOM) Level 3 Core Specification. (April):1–216, 2004.

A

IMPLEMENTED @FIELDS FOR LUCENE INDEXING

This Annex has tables with the column names in data persistence classes with the name of the *@Fields* declared for *Lucene* indexation.

Columns	Fields
qu_query_date	• quDateSort
qu_keywords	• q_keywordsCS • q_keywordsNCS
qu_organism	• q_organismCS • q_organismNCS
qu_matching_publications	• quMatchingPublicationsSort
qu_query_name	• q_query_nameCS • q_query_nameNCS • quNameSort

Table 13.: *Queries' @Fields*

Columns	Fields
crp_corpus_name	• crp_nameCS • crp_nameNCS • crpNameSort
crp_notes	• crp_notesCS • crp_notesNCS • crpNotesSort

Table 14.: *Corpus' @Fields*

Columns	Fields
au_username	• auUsernameCS • auUsernameNCS • auUsernameSort
au_fullname	• auFullNameCS • auFullNameNCS • auFullNameSort
au_email	• auEmailCS • auEmailNCS • auEmailSort
au_location	• auLocationCS • auLocationNCS • auLocationSort

Table 15.: *AuthUsers' @Fields*

Columns	Fields
pro_name	• pro_nameCS • pro_nameNCS • proNameSort
pro_notes	• pro_notesCS • pro_notesNCS

Table 16.: *Processes' @Fields*

Columns	Fields
po_description	• pro_po_descriptionCS • pro_po_descriptionNCS • pro_po_descriptionSort

Table 17.: *ProcessOrigins' @Fields*

Columns	Fields
pt_process_type	• pro_pt_typeCS • pro_pt_typeNCS • pro_pt_typeSort

Table 18.: *ProcessTypes' @Fields*

Columns	Fields
pub_title	• titleCS • titleNCS • pubTitleSort
pub_authors	• authorsCS • authorsNCS • pubAuthorsSort
pub_category	• categoryCS • categoryNCS • pubCategorySort
pub_journal	• journalCS • journalNCS
pub_abstract	• abstractCS • abstractNCS
pub_fullcontent	• fullContentCS • fullContentNCS
pub_notes	• notesCS • notesNCS
pub_type	• typeCS • typeNCS • pubTypeSort

Table 19.: *Publications' @Fields*

Columns	Fields
res_element	• keywordEdgeNGram_res_element • tokenEdgeNGram_res_element • res_elementCS • res_elementNCS

Table 20.: *ResourceElements' @Fields*

Columns	Fields
reso_resource_name	• reso_resource_nameCS • reso_resource_nameNCS • resoNameSort
reso_notes	• reso_notesCS • reso_notesNCS • resoNotesSort

Table 21.: *Resources' @Fields*

Columns	Fields
syn_synonym	• keywordEdgeNGram_syn_synonym • tokenEdgeNGram_syn_synonym • syn_synonymCS • syn_synonymNCS

Table 22.: *SynonymsId's @Fields*