BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

INDEXING OF BIG TEXT DATA AND SEARCHING IN THE INDEXED DATA

INDEXACE ROZSÁHLÝCH TEXTOVÝCH DAT A VYHLEDÁVÁNÍ V ZAINDEXOVANÝCH DATECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. DAVID KOZÁK

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. JAROSLAV DYTRYCH, Ph.D.

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2019/2020

Master's Thesis Specification

22961

Student:	**Kozák David, Bc.**
Programme:	Information Technology Field of study: Information Systems
Title:	**Indexing of Big Text Data and Searching in the Indexed Data**
Category:	Web

Assignment:

1. Study indexing approaches and tools suitable for big data.
2. Get acquainted with MG4J tool and with semantically enriched data available in the Knowledge Technology Research Group at FIT BUT.
3. Design a distributed system for indexing large textual data and semantic querying over the data. Focus on defining appropriate interfaces between system components and on the stability of the system as a whole.
4. Implement the designed solution and perform tests over the real big data.
5. Evaluate your work and create a brief poster presenting it.

Recommended literature:

- According to the supervisor's recommendation

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Dytrych Jaroslav, Ing., Ph.D.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 20, 2020
Approval date:	November 1, 2019

Abstract

The topic of this thesis is semantic searching over big textual data. The goal is to design and implement a search engine that queries the semantically enhanced documents efficiently and has a user friendly interface for working with the results. Firstly, state of the art solutions along with their strengths and shortcomings are analyzed. Then a design for new search engine is presented along with a specialized query language. The system consists of components for indexing and searching the documents, management server, compiler for the query language and two clients, web based and command line. The engine has been successfully designed, developed and deployed and is available via the Internet. As a result of that, the possibility of using of the semantic searching is available to a wide audience.

Abstrakt

Tématem této práce je sémantické vyhledávání ve velkých textových datech. Cílem je navrhnout a implementovat vyhledávač, který se bude efektivně dotazovat nad sémanticky obohacenými dokumenty a prezentovat výsledky uživatelsky přívětivým způsobem. V práci jsou nejdříve analyzovány současné sémantické vyhledávače, spolu s jejich silnými a slabými stránkami. Poté je přednesen návrh nového vyhledávače s vlastním dotazovacím jazykem. Tento systém se skládá z komponent pro indexaci a dotazování se nad dokumenty, management serveru, překladače pro dotazovací jazyk a dvou klientských aplikací, webové a konzolové. Vyhledávač byl úspěšně navržen, implementován i nasazen a je veřejně dostupný na Internetu. Výsledky práce umožňují široké veřejnosti využívat sémantického vyhledávání.

Keywords

search engine, semantic enhancement, MG4J, compiler, indexation, searching, annotation, big data

Klíčová slova

vyhledávač, sémanticky obohacené texty, MG4J, překladač, indexace, vyhledávání, anotace, big data

Reference

KOZÁK, David. *Indexing of Big Text Data and Searching in the Indexed Data.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jaroslav Dytrych, Ph.D.

Rozšířený abstrakt

Tématem této práce je sémantické vyhledávání ve velkých textových datech. Výzkumná skupina znalostních technologií (KNOT – Knowledge Technology Research Group) na Fakultě informačních technologií Vysokého učení technického v Brně disponuje skupinou nástrojů pro zpracování přirozeného jazyka, která umožňuje analyzovat dokumenty psané v přirozených jazycích a přidávat k nim další metainformace. Tyto informace mohou být buď syntaktické, jako například lemma slov a jejich pozice ve větě a odstavci, či sémantické, například entity typu člověk či místo. Každá z těchto entit má svoji vlastní sadu atributů dále rozšiřujících kontext dokumentu. Například u entity typu osoba nalezneme atributy jméno, datum narození apod. Výstupem tohoto zpracování je velké množství textových dat. Řádově mluvíme o milionech dokumentů zabírajících stovky GB místa na disku. Už samo o sobě se jedná o velký kus dobře odvedené práce, nicméně tyto dokumenty jsou spíše vhodné pro automatické zpracování než pro čtení člověkem. Navíc je jich takové množství, že bez možnosti v těchto sémanticky obohacených datech rychle vyhledávat je jejich použití omezeno. Cílem této práce je navrhnout a implementovat vyhledávač, který by se efektivně dotazoval nad dokumenty a zároveň umožňoval využít v dotazech všechny dostupné metainformace.

Několik vyhledávačů s podporou pro sémantické vyhledávání jako například Mimir či Sketch Engine již bylo implementováno dříve. Jeden takový vyhledávač byl dokonce vyvinut interně uvnitř KNOT. Nicméně, žádný z nich nesplňoval požadavky kladené na nový vyhledávač. Některé vyhledávače nepodporovaly entity s atributy, jiné byly zase až příliš komplexní, zahrnující příliš mnoho dalších služeb, které pro tento účel nebyly potřebné. Jejich využití by proto bylo těžkopádné. Vyhledávač dříve implementovaný členy KNOT splňoval požadavky nejlépe, naneštěstí ale nebyl stabilní a jeho kód byl těžko udržovatelný. Proto bylo rozhodnuto, že se vytvoří nový vyhledávač, ve kterém bude kladen důraz právě na stabilitu a udržovatelnost.

V práci byl nejdříve proveden důkladný návrh vyhledávače jakožto distribuovaného systému skládajícího se z komponent různých typů. Pro indexování a přípravu dat byla navržena komponenta *IndexBuilder*, pro vyhodnocení dotazu komponenta *IndexServer*. Tyto komponenty interně využívají pro indexování dokumentů *MG4J – Managening Gigabytes for Java*, proto v práci naleznete i sekci diskutující tuto knihovnu. Dotazy mohou přijít ze dvou různých klientů. Prvním z nich je *WebClient*, tvořící primární uživatelské rozhraní systému. Druhým je *ConsoleClient*, sloužící pro testovaní a automatizované dotazování. Tyto komponenty jsou tvořeny z komplexní hierarchie modulů umožňujících efektivně znovuvyužívat jejich funkcionalitu. V práci byly také navrženy čtyři různé datové struktury pro přenos anotovaného textu mezi komponentami.

Pro dotazování nad sémanticky obohacenými texty je třeba speciálního dotazovacího jazyka. Tento jazyk by měl být dostatečně expresivní, aby umožňoval dotazovat se s pomocí všech dostupných metadat, ale zároveň by měl být jednoduchý na pochopení, aby s ním byli schopní pracovat i lidé z jiných domén než informační technologie. Jako součást této práce byl vyvinut jazyk *Enticing Query Language (EQL)*, který by měl splňovat výše uvedené požadavky. *EQL* rozšiřuje sémantiku vyhledávacího jazyka knihovny *MG4J* o dotazování se nad entitami s atributy, globální omezení pro definici vztahů mezi entitami a omezení vyhledávání na konkrétní dokument. Vzhledem k širší sémantice bylo v rámci práce nutné navrhnout vlastní vyhledávací algoritmy, které budou tato dodatečná omezení schopny vyhodnotit. Tímto byla také otevřena cesta k úpravě způsobu vyhodnocení dotazů tak, aby byly vráceny všechny kombinace výsledků, které se v dokumentu nacházejí. Překladač tohoto jazyka byl plně integrován do infrastruktury vyhledávače.

Vyhledávač byl úspěšně navržen, implementován, testován a nasazen do provozu. Jádro projektu bylo dále rozšířeno o monitorovací a administrativní infrastrukturu, konfigurační doménově specifický jazyk, podporu pro asynchronní načítání výsledků a inteligentní vyhledávací řádek zobrazující výsledky syntatických a sémantických kontrol. Platforma může být dále rozšířena například přidáváním nových typů indexačních serverů, nativním mobilním klientem či podporou pro dědičnost mezi entitami.

Indexing of Big Text Data and Searching in the Indexed Data

Declaration

I hereby declare that this thesis was prepared as an original work by the author under the supervision of Ing. Jaroslav Dytrych, Ph.D. The supplementary information was provided by Doc. RNDr. Pavel Smrž, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this project.

. . . .
David Kozák
June 2, 2020

Acknowledgements

I would like to thank my supervisor, Ing. Jaroslav Dytrych, Ph.D., and Doc. RNDr. Pavel Smrž, Ph.D for their professional help and guidance. I also want to thank Ing. Jan Doležal for his guidance and support in the design process. Last but not least, I would like to thank my parents for their neverending support.

Contents

Chapter 1

Introduction

The topic of this thesis is semantic searching over big textual data. The Knowledge Technology Research Group (KNOT)[1] at the Faculty of Information Technology Brno University of Technology (FIT BUT) has a Natural language processing (NLP) pipeline which can analyse documents written in natural languages and add additional meta information to them. Such information can be *syntactic*, such as lemma of the word or their position within sentences and paragraphs, or *semantic*, such as entities like people and places. The output of this pipeline is a big volume of textual data. It is already a great piece of work on its own, but without the ability to query these semantically enhanced documents, their usage is limited. The goal of this thesis is to design and develop a search engine that would query the documents efficiently while allowing to use all the meta information in the queries.

A couple of search engines with support for semantic search such as Mimir or Sketch Engine have been implemented before. One such engine has even been created internally within KNOT[2]. However, none of those matched the requirements for the new engine. They either did not provide support for entities with attributes or were way too complex for this use case. The previous engine developed at KNOT was quite close, but unfortunately it wasn't stable and the code was hard to maintain. That's why the decision was made to create a new engine and design it with stability and maintainability in mind.

In order to query the semantic metadata, a special query language has to be used. This language should be powerful enough to query all the entities inside semantically enhanced documents but it should also be simple to understand, so that users from other domains can use it as well. As a part of this thesis, such query language called EQL has been designed and its compiler was integrated into the search engine infrastructure.

The structure of the text is as follows. The problem of searching and indexing is described in the chapter 2. The process of semantic enhancement and its realization via the Corpora processing pipeline is presented in the chapter 3. Various state of the art semantic search engines along with their strengths and shortcomings are analyzed as well. The chapter 4 presents the design of the new search engine called Enticing. The chapter 5 describes languages, libraries and frameworks used when developing Enticing. The chapter 6 contains more in-depth information about the platform. The chapter 7 describes how the search engine was tested. The chapter 8 covers the deployment process. In the end, the conclusion is given.

[1]https://www.fit.vut.cz/research/group/knot/
[2]http://knot.fit.vutbr.cz/projects.html

Chapter 2

Indexing and searching inside search engines

This chapter describes the problem of indexing and searching in the context of search engines. The section 2.1 contains basic definitions used in the rest of the document. Typical techniques used within search engines are described in 2.2. A search engine called MG4J is presented in 2.3, as it is used internally within Enticing.

2.1 Basic definitions

In this section, the basic terms used throughout the text are defined. Since this thesis is a follow-up work of [14, 8], the terminology will be mostly identical as it was defined in [14] with some modifications and extensions.

Token

Token is a commonly used term in scientific texts and it's definition differs based on the domain. For the purposes of this thesis, it can be defined as a sequence of non-whitespace characters such as letters, numbers and special characters.

Index

The Oxford dictionary defines it as an alphabetical list of names, subjects, etc. with reference to the pages on which they are mentioned[1].

In [14], they defined index as a structure allowing a faster access to a certain piece of information without the need to process all the data. This definition is well suited for this thesis, so it will be adopted.

Indexing

Indexing can be defined as creating tables (indexes) that point to the location of folders, files and records. Depending on the purpose, indexing identifies the location of resources based on file names, key data fields in a database record, text within a file or unique attributes in a graphics or video file[2]. For our purposes, indexing will be used to describe the process

[1] `https://www.lexico.com/en/definition/index`
[2] `https://www.pcmag.com/encyclopedia/term/44896/indexing`

of creating all the metadata necessary to query the semanticaly enhanced documents. In Enticing, indexing is performed as a preprocessing step before the services are started.

Searching

Searching can be defined as the process of looking up a certain piece of information using a query. In this thesis, that piece of information will be snippets of texts from documents and the query will be written in EQL.

Snippet

If a query matched a document, the result has to be presented to the user. One of the most common forms of presenting search results are snippets. A snippet is a part of a document that matched given query, possibly extended with some additional information about the evaluation of the searching algorithm.

2.2 Techniques used in a search engine

This section focuses on some of the underlying principles, algorithms and data structures that are used inside state of the art search engines, which are relevant for our purposes.

Inverted Indexes

An inverted index over a collection of documents contains, for each term of the collection, the set of documents in which the term appears and additional information such as the number of occurrences of the term within each document, and possibly their positions [18]. This data structure is used inside search engines to efficiently determine which documents contain the specified words.

Query expansion

Modern web search engines rely on query expansion, an automatic or semi-automatic mechanism that aims to rewrite the user intent (i.e., a set of keywords, maybe with additional context such as geographical location, past search history, etc.) as a structured query built upon a number of operators [5].

Tree based evaluation of the query

One of the most used data structures for representing a search query is a tree [14]. The leaves represent keywords from the query and the intermediary nodes represent operators. The searching algorithm can then proceed in the bottom up way as follows. First, all the leaves of the tree are evaluated and the results are stored within them. Then their parents are evaluated, combining results from their children. The execution proceeds all the way to the root, which represents the whole query.

Semantic models of searching

The semantic of the structured query is given by the semantic model. The simplest one is the boolean model, where only conjuctions, disjunctions, negations and keywords are allowed. Unfortunately, this model does not provide any information regarding the fact how the document was matched by the query. MG4J uses a different model, which is called Minimal Interval Semantics. It uses intervals of natural numbers that are incomparable towards inclusion to represent the semantics of a query. Each interval is a witness of the satisfiability of the query, and defines a region of the document that satisfies the query [5]. After the bottom up algorithm finishes, all intervals that are stored within the root of the query represent a successful match.

Parallel execution

Since the amount of data that has to be processed for every query is huge, it is useful to distribute the data to multiple servers (and possibly multiple collections on each server) to paralelize the process. However, one must find the optimal degree of distribution, because the cost of combining the results might eventually overcome the speed gain. And even before the querying itself, the process of creating the indexes is time-consuming, therefore it is better to split the inputs and do it in parallel.

2.3 MG4J – Managing Gigabytes for Java

This section introduces MG4J, a free full-text search engine for large documents written in Java [4]. It is developed under the GNU Lesser General Public License[3] at the University degli Studi di Milano[4]. MG4J is used internally inside Enticing. The engine and the research around it is an extensive topic that does not fit into the scope of one section. Therefore only basic introduction and parts relevant for this thesis are covered. For additional information, please refer to the manual [3].

MG4J has a query language with very expressive set of operators allowing to build complex queries. These operators are implemented using new very efficient search algorithms [3]. It supports searching over multiple indexes and combining the results. On top of that, MG4J is open source, so it is possible to dive into the source code when the answers cannot be found in the documentation. Unfortunately, it has no support for entities with attributes and relationships between them. Nevertheless, the aforementioned properties make it a very suitable backend for Enticing.

Indexing in MG4J works in a sequence of steps. The first one is scanning documents and creating batches. Firstly, each document is given an identifier, as can be seen in the table 2.1. Afterwards, all words within documents are given identifiers, which can be seen in the table 2.2. From two previous steps, triples (v, p, d) are created, where v stands for an id of a word, d is an id of the document and p is a position within document. Example of these triplets is given in the table 2.3. The inverted index is then created by sorting these triplets based on the id of the word, which you can see in the table 2.4. The resulting batches have to be merged again to create a full index [8].

[3]https://www.gnu.org/licenses/lgpl-3.0.en.html
[4]http://www.unimi.it/

Identifier	Document
0	I love you
1	God is love
2	Love is blind
3	Blind justice

Table 2.1: Document identifiers

Identifier	Word
0	blind
1	god
2	i
3	is
4	justice
5	love
6	you

Table 2.2: Word identifiers

Triples	Word	Document
(2,0,0)	i	I love you
(5,0,1)	love	I love you
(6,0,2)	you	I love you
(1,1,0)	god	God is love
(3,1,1)	is	God is love
(5,1,2)	love	God is love
(5,2,0)	love	Love is blind
(3,2,1)	is	Love is blind
(0,2,2)	blind	Love is blind
(0,3,0)	blind	Blind justice
(4,3,1)	justice	Blind justice

Table 2.3: Triplets before sorting

Word	Locations
0 (blind)	(0,2,2), (0,3,0)
1 (god)	(1,1,0)
2 (i)	(2,0,0)
3 (is)	(3,1,1), (3,2,1)
4 (justice)	(4,3,1)
5 (love)	(5,0,1), (5,1,2), (5,2,0)
6 (you)	(6,0,2)

Table 2.4: Inverted index

Chapter 3

Semantic enhancement of natural languages

This chapter explains the topic of semantic enhancement. Related basic definitions are provided in 3.1. The section 3.2 describes how semantically enhanced documents are created within KNOT. A comparison of three state of the art search engines with support for semantic search is given in 3.3.

3.1 Basic definitions

The key definition in semantic enhancement is **Semantic annotation**. Semantic annotations are metadata assigned to other data in order to increase their context and semantics [14]. These annotations are usually derived from unstructured content using Natural language processing and afterwards they are encoded in a structured format suitable for semantic search [16].

Semantic search over documents aims to find information that is not based just on the presence of words, but also on their meaning. It is gradually establishing itself as the next generation search paradigm, which can better satisfy a wider range of information needs, as compared to traditional full-text search. In the case of semantic search, what is being indexed is typically a combination of words, formal knowledge typically expressed in an ontology, and semantic annotations mentioning ontological concepts in the text [16].

3.2 Corpora processing tools

This section describes the corpora processing pipeline[1] which is used within KNOT to create semantically enhanced documents.

Input data

In [14, 8], the input data was CommonCrawl and Wikipedia. The English wikipedia publishes a dump[2] of it's whole database every month, which can be used for various analysis, statistic measurements, etc.

[1] http://knot.fit.vutbr.cz/corpproc/corpproc_en.html
[2] https://dumps.wikimedia.org/

CommonCrawl[3]is a project that maintains an open repository of web crawl data.

For the practical part of this thesis, the data from Wikipedia was chosen, but in fact any source of input data in the mg4j format can be used. The details of this format are explained in the following subsection.

Stages of the pipeline

This section describes the stages of the processing pipeline along with their inputs and outputs.

Verticalization

The input of verticalization can be *a warc.gz* archive, a dump of Wikipedia or a webpage in the HTML format. A meaningful text is extracted from web pages using the tool Justext[4]. During this step the HTML pages are parsed. The output of verticalization is a file with three columns, where each line describes one token. It still contains some XML tags, but it is not a valid XML document anymore. The tags only serve as metainformation about the text [8].

Deduplication

As the name suggests, the goal of deduplication is to remove duplicit pages and duplicit pieces of text within pages [8]. The tool used for deduplication was inspired by a tool called Onion[5]. The deduplication consists of two phases. In the first the URL of the document is used to detect duplicit documents. If the URL is unique, the hash of the text of the document is computed and compared with hashes of previously processed documents. If the document passes this test, it is assumed to be unique. Then, the second stage is performed in which the content of the document is checked for duplicities, again using hashes. The output of this stage are files in the same format, without the duplicities.

Tagging

In this stage a tool TreeTagger[6] is used to identify parts of speech. The goal is to identify the type of each token using a morphological analysis [8].

Parsing

This stage performs a syntactical analysis using a tool called MDParser[7]. Tokens are enhanced with various syntactical information.

SEC

In the end, a tool called SEC is used, which identifies entities within documents. It was developed by Jan Doležal in his Master's thesis [7]. The output of this stage are files in the mg4j format described in the subsection 3.2. Some of the most frequently used searches

[3]https://commoncrawl.org/
[4]http://corpus.tools/wiki/Justext
[5]http://corpus.tools/wiki/Onion
[6]http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/
[7]http://mdparser.sb.dfki.de/

are for people, locations, organisations, and other named entities [16]. Therefore this step provides very useful pieces of information.

Mg4j file format

This subsection describes the mg4j file format, which is the output of the Corpora processing pipeline. It is a token separated value (tsv) format, where each column has a well defined meaning. It's current format is defined in the table 3.1.

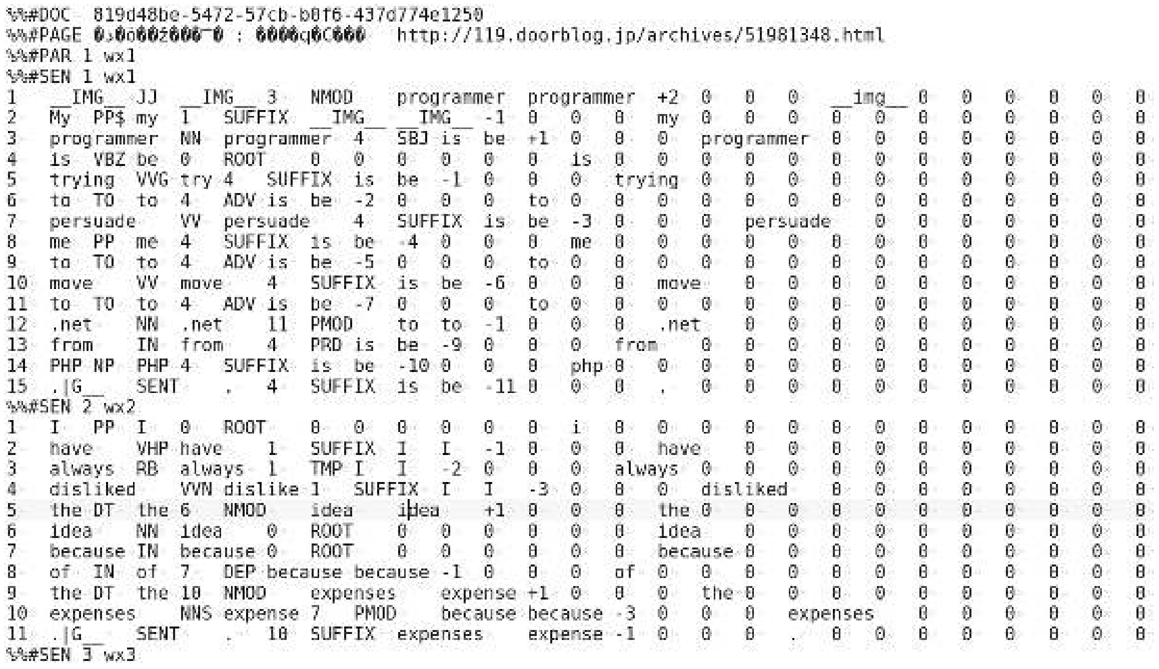
Number	Name	Description
1	position	position within sentence
2	token	the token itself
3	tag	token type
4	lemma	basic shape of the token
5	parpos	token connected to current token
6	function	function of the token within sentence
7	parword	value of token identified by parpos
8	parlemma	the basic shape of the parword
9	paroffset	offset between token and parpos
10	link	source of the data
11	length	number of tokens connected with this token
12	docuri	uri of the document
13	lower	token in lowercase
14	nerid	id of the entity
15	nertag	type of the entity
16-25	param0 – param9	entity attributes
26	nertype	type of SEC/NER
27	nerlength	length of the entity

Table 3.1: MG4J File format

Columns 16-25 are polymorphic. Their meaning differs based on the entity, which is determined by the value in the nertag column. The exact meaning of the attribute columns as of writing this section is defined in the table 3.2, but please note that the types of entities and their attributes change over time.

entity	param0	param1	param2	param3	param4	param5	param6	param7	param8	param9
person	url	image	name	gender	birthplace	birthdate	deathplace	deathdate	profession	nationality
artist	url	image	name	gender	birthplace	birthdate	deathplace	deathdate	role	nationality
location	url	image	name	country						
artwork	url	image	name	form	datebegun	datecompleted	movement	genre	author	
event	url	image	name	startdate	enddate	location				
museum	url	image	name	type	established	director	location			
family	url	image	name	role	nationality	members				
group	url	image	name	role	nationality					
nationality	url	image	name	country						
date	url	image	year	month	day					
interval	url	image	fromyear	frommonth	fromday	toyear	tomonth	today		
form	url	image	name							
medium	url	image	name							
mythology	url	image	name							
movement	url	image	name							
genre	url	image	name							

Table 3.2: Currently used entities and their attributes



```
%%#DOC  819d48be-5472-57cb-b0f6-437d774e1250
%%#PAGE 0¸0ó00ž000¯0 : 0000q0C000   http://119.doorblog.jp/archives/51981348.html
%%#PAR 1 wx1
%%#SEN 1 wx1
1   __IMG__ JJ  __IMG__  3   NMOD    programmer  programmer  +2  0   0   0   __img__ 0   0   0   0   0   0
2   My  PP$ my  1   SUFFIX  __IMG__ __IMG__ -1  0   0   0   my  0   0   0   0   0   0   0   0   0   0   0
3   programmer  NN  programmer  4   SBJ is  be  +1  0   0   0   programmer  0   0   0   0   0   0   0   0
4   is  VBZ be  0   ROOT    0   0   0   0   0   0   is  0   0   0   0   0   0   0   0   0   0   0   0   0
5   trying  VVG try 4   SUFFIX  is  be  -1  0   0   0   trying  0   0   0   0   0   0   0   0   0   0   0
6   to  TO  to  4   ADV is  be  -2  0   0   0   to  0   0   0   0   0   0   0   0   0   0   0   0   0   0
7   persuade    VV  persuade    4   SUFFIX  is  be  -3  0   0   0   persuade    0   0   0   0   0   0   0
8   me  PP  me  4   SUFFIX  is  be  -4  0   0   0   me  0   0   0   0   0   0   0   0   0   0   0   0   0
9   to  TO  to  4   ADV is  be  -5  0   0   0   to  0   0   0   0   0   0   0   0   0   0   0   0   0   0
10  move    VV  move    4   SUFFIX  is  be  -6  0   0   0   move    0   0   0   0   0   0   0   0   0   0
11  to  TO  to  4   ADV is  be  -7  0   0   0   to  0   0   0   0   0   0   0   0   0   0   0   0   0   0
12  .net    NN  .net    11  PMOD    to  to  -1  0   0   0   .net    0   0   0   0   0   0   0   0   0   0
13  from    IN  from    4   PRD is  be  -9  0   0   0   from    0   0   0   0   0   0   0   0   0   0   0
14  PHP NP  PHP 4   SUFFIX  is  be  -10 0   0   0   php 0   0   0   0   0   0   0   0   0   0   0   0   0
15  .|G__   SENT    .   4   SUFFIX  is  be  -11 0   0   0   .   0   0   0   0   0   0   0   0   0   0   0
%%#SEN 2 wx2
1   I   PP  I   0   ROOT    0   0   0   0   0   0   i   0   0   0   0   0   0   0   0   0   0   0   0   0
2   have    VHP have    1   SUFFIX  I   I   -1  0   0   0   have    0   0   0   0   0   0   0   0   0   0
3   always  RB  always  1   TMP I   I   -2  0   0   0   always  0   0   0   0   0   0   0   0   0   0   0
4   disliked    VVN dislike 1   SUFFIX  I   I   -3  0   0   0   disliked    0   0   0   0   0   0   0   0
5   the DT  the 6   NMOD    idea    idea    +1  0   0   0   the 0   0   0   0   0   0   0   0   0   0   0
6   idea    NN  idea    0   ROOT    0   0   0   0   0   0   idea    0   0   0   0   0   0   0   0   0   0
7   because IN  because 0   ROOT    0   0   0   0   0   0   because 0   0   0   0   0   0   0   0   0   0
8   of  IN  of  7   DEP because because -1  0   0   0   of  0   0   0   0   0   0   0   0   0   0   0   0
9   the DT  the 10  NMOD    expenses    expense +1  0   0   0   the 0   0   0   0   0   0   0   0   0   0
10  expenses    NNS expense 7   PMOD    because because -3  0   0   0   expenses    0   0   0   0   0   0
11  .|G__   SENT    .   10  SUFFIX  expenses    expense -1  0   0   0   .   0   0   0   0   0   0   0   0
%%#SEN 3 wx3
```

Figure 3.1: **Example of an mg4j file**

An example of an mg4j file can be seen in the figure 3.1.

3.3 State of the art semantic search engines

This section covers three state of the art search engines which support semantic search to a various extent. Their strengths and shortcomings are analyzed, as these engines were used as inspiration when designing Enticing.

Mimir

Mimir is a semantic search engine developed at the The University of Sheffield. It was developed as a part of the Gate infrastructure for language engineering[8]. Using index federation and cloud-based deployment, it can scale up to 150 millions documents [16]. It supports hybrid queries that arbitrarily mix full-text, structural, linguistic and semantic constraints [16]. It internally combines different indexing technologies. The full-text search is done using MG4J and a triple store queried using SPARQL is then used for accessing Linked Open Data resources. The overview can be seen in the figure 3.2.

Another interesting feature of Mimir is that it uses direct indexes in addition to the widely used inverted indexes, in order to support both information discovery and information seeking searches [16].

Queries in Mimir are trees, with compound query operators as intermediary nodes and base queries as leaves. They are evaluated using a bottom up algorithm. During the evaluation of the query, token query executors are created for each leaf. They gather all the hits in the document. Then their parents are processed, performing compound operations

[8]https://gate.ac.uk/

on the intervals obtained by their children. This process is carried up all the way to the root of the tree, which then contains intervals that match the whole query [16].

Mimir is a very powerful engine, even too powerful for our use case. Most of its functionality is not necessary, therefore using it would be too heavyweight.

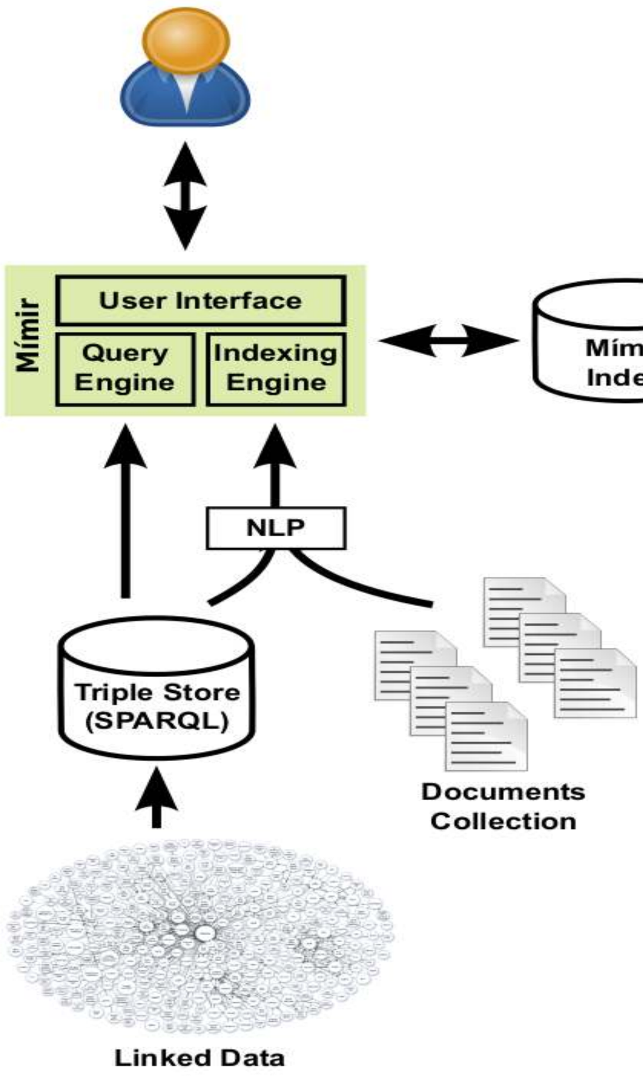


Figure 3.2: **Mimir overview** Taken from [16]

Sketch Engine

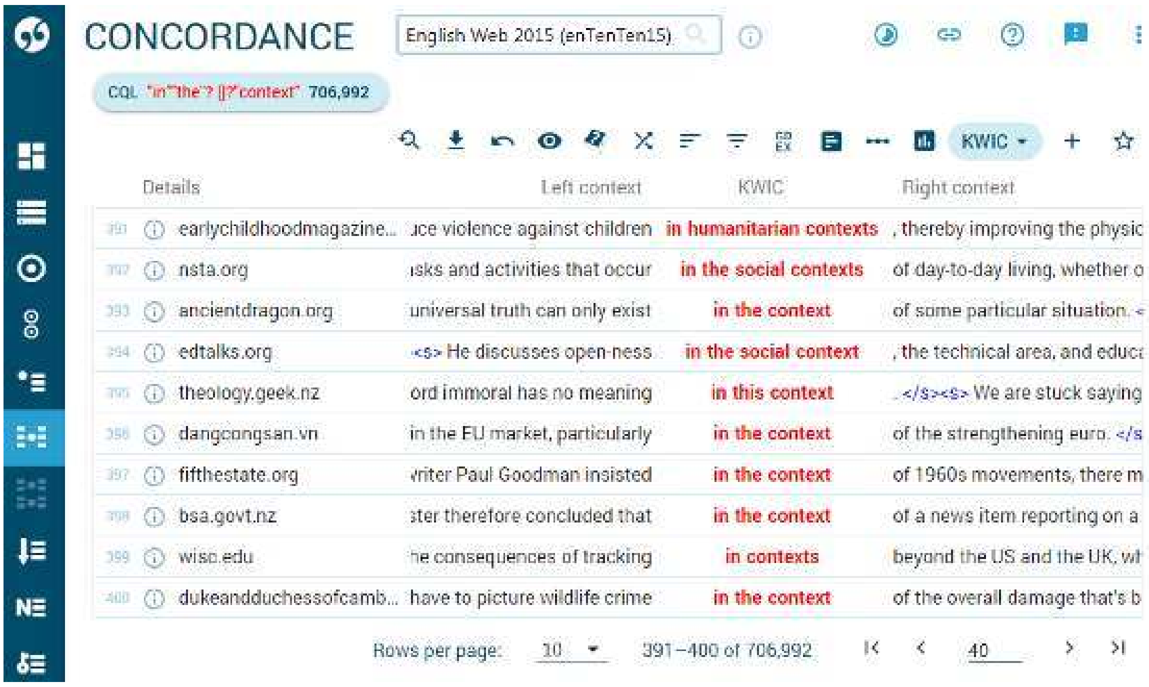
Sketch Engine is a tool for analyzing how languages work. It analyses large text corpuses in order to find out what is typical and what is rare for a given language. It houses more than 500 corpora in more than 90 languages [12].

It presents search results in three forms, word sketches, concordances or word lists.

The word sketch processes the word's collocates and other words in its surroundings. The results are organized into categories, called grammatical relations such as words that serve as an object of the verb, words that serve as a subject of the verb, words that modify the word [12].

Word lists serve display the frequency of each word in the documents.

From our perspective, the most interesting form of presenting the results are the concordances. In this mode we can search words, phrases, tags, documents, etc. and display them along with their context. An example can be seen in the figure 3.3.


CONCORDANCE

English Web 2015 (enTenTen15)

CQL "in""the"? []?"context" 706,992

KWIC

	Details	Left context	KWIC	Right context
391	earlychildhoodmagazine...	ıce violence against children	in humanitarian contexts	, thereby improving the physic
392	nsta.org	ısks and activities that occur	in the social contexts	of day-to-day living, whether o
393	ancientdragon.org	universal truth can only exist	in the context	of some particular situation. <
394	edtalks.org	<s> He discusses open-ness	in the social context	, the technical area, and educ
395	theology.geek.nz	ord immoral has no meaning	in this context	. </s><s> We are stuck saying
396	dangcongsan.vn	in the EU market, particularly	in the context	of the strengthening euro. </s
397	fifthestate.org	vriter Paul Goodman insisted	in the context	of 1960s movements, there m
398	bsa.govt.nz	ster therefore concluded that	in the context	of a news item reporting on a
399	wisc.edu	he consequences of tracking	in contexts	beyond the US and the UK, wl
400	dukeandduchessofcamb...	have to picture wildlife crime	in the context	of the overall damage that's b

Rows per page: 10 391–400 of 706,992 40

Figure 3.3: **SketchEngine** Concordance view

Even though SketchEngine is an interesting piece of work that can be used for studying languages, it was not designed to be a publicly available search engine. On the other hand, some of its features, mainly the concordances, can be used by our search engine as well.

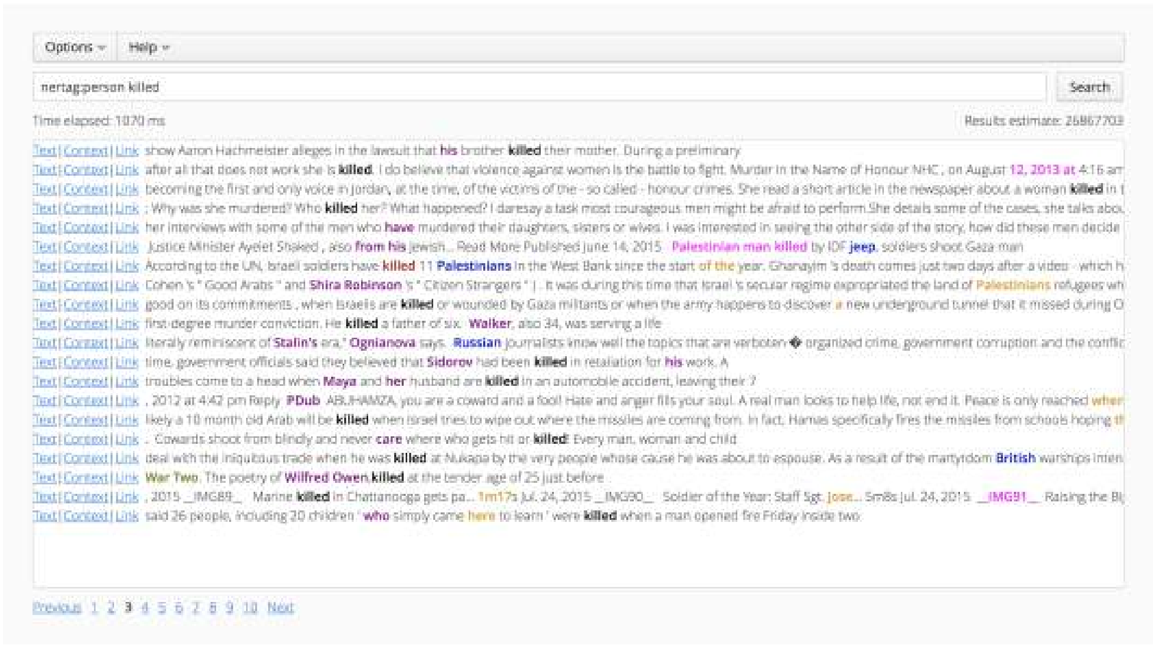
Previous system at FIT BUT

One semantic search engine was developed within KNOT. Its original author was Jan Kouril[9] and it was extended by Sergey Panov in [14] and Katarina Gresova in [8]. Its architecture was the following. There were two main components, *Webserver* and *IndexDeamon*. *IndexDeamon* was able to index documents and query them. *Webserver* provided user interface for the system. You can find a screenshot of the old user interface in the figure 3.4. It was taken from [8] as the system is no longer in use.

For querying the documents, package Query along with a query language mg4j-eql was developed in [14]. The Query package was created so that it could be used in various clients. It was later integrated into the old webserver in [8]. K. Gresova also made several changes to the webserver to make it more flexible. However, the original code was not written in a maintainable way. And, as it typically happens with software products, its quality got only worse over time. On top of that, its implementation was not fully finished, because

[9]https://www.fit.vut.cz/person/ikouril/

it was out of the scope for a bachelor thesis. And because of the maintainability issues, it was hard to add the missing pieces of functionality or provide bug fixes. However, a lot of ideas behind the old design could be reused in Enticing.



```
nertag:person killed
Time elapsed: 1070 ms                                                    Results estimate: 26867703
Text | Context | Link  show Aaron Hachmeister alleges in the lawsuit that his brother killed their mother. During a preliminary
Text | Context | Link  after all that does not work she is killed. I do believe that violence against women is the battle to fight. Murder in the Name of Honour NHC , on August 12, 2013 at 4:16 am
Text | Context | Link  becoming the first and only voice in Jordan, at the time, of the victims of the - so called - honour crimes. She read a short article in the newspaper about a woman killed in t
Text | Context | Link  ; Why was she murdered? Who killed her? What happened? I daresay a task most courageous men might be afraid to perform.She details some of the cases, she talks abou
Text | Context | Link  her interviews with some of the men who have murdered their daughters, sisters or wives. I was interested in seeing the other side of the story, how did these men decide
Text | Context | Link  Justice Minister Ayelet Shaked , also from his Jewish... Read More Published June 14, 2015  Palestinian man killed by IDF jeep, soldiers shoot Gaza man
Text | Context | Link  According to the UN, Israeli soldiers have killed 11 Palestinians in the West Bank since the start of the year. Ghanayim 's death comes just two days after a video - which h
Text | Context | Link  Cohen 's " Good Arabs " and Shira Robinson 's " Citizen Strangers " ) . It was during this time that Israel 's secular regime expropriated the land of Palestinians refugees wh
Text | Context | Link  good on its commitments , when Israelis are killed or wounded by Gaza militants or when the army happens to discover a new underground tunnel that it missed during O
Text | Context | Link  first-degree murder conviction. He killed a father of six.  Walker, also 34, was serving a life
Text | Context | Link  literally reminiscent of Stalin's era," Ognianova says.  Russian journalists know well the topics that are verboten � organized crime, government corruption and the conflic
Text | Context | Link  time, government officials said they believed that Sidorov had been killed in retaliation for his work. A
Text | Context | Link  troubles come to a head when Maya and her husband are killed in an automobile accident, leaving their ?
Text | Context | Link  , 2012 at 4:42 pm Reply  PDub  ABUHAMZA, you are a coward and a fool! Hate and anger fills your soul. A real man looks to help life, not end it. Peace is only reached when
Text | Context | Link  likely a 10 month old Arab will be killed when Israel tries to wipe out where the missiles are coming from. In fact, Hamas specifically fires the missiles from schools hoping th
Text | Context | Link  .  Cowards shoot from blindly and never care where who gets hit or killed! Every man, woman and child
Text | Context | Link  deal with the iniquitous trade when he was killed at Nukapa by the very people whose cause he was about to espouse. As a result of the martyrdom British warships inten
Text | Context | Link  War Two. The poetry of Wilfred Owen,killed at the tender age of 25 just before
Text | Context | Link  , 2015 _IMG89_   Marine killed in Chattanooga gets pa... 1m17s Jul. 24, 2015 _IMG90_   Soldier of the Year: Staff Sgt. Jose... 5m8s Jul. 24, 2015  _IMG91_  Raising the Bi
Text | Context | Link  said 26 people, including 20 children ' who simply came here to learn ' were killed when a man opened fire Friday inside two

Previous  1  2  3  4  5  6  7  8  9  10  Next
```

Figure 3.4: **UI of the previous solution** Taken from [8]

Chapter 4

Design of the search engine

This chapter covers the design of the search engine. The section 4.1 contains the requirement analysis. The design of EQL is described in 4.2. In the section 4.3, components of the platform are presented. The architecture of individual modules is given in 4.4. The section 4.5 covers the data structures designed for representing and transferring the annotated text.

4.1 Requirements analysis

This section contains the requirements analysis. Firstly, an informal specification is given. Afterwards, a use case diagram is presented and described.

Search engine specification

Enticing is a search engine for querying semantically enhanced documents available in the Knowledge Technology Research Group. Users are provided with a special query language called Enticing Query Language (EQL) allowing them to query not just the text of the documents, but also metadata assigned to it. Such metadata can be *syntactic*, such as lemmas of the words, parts of speech, positions within sentences and paragraphs, or *semantic*, such as entities like people, places and events. These entities may have various attributes. For example person might have attributes like name and birthdate. All these attributes can be used inside queries. The exact format of metadata is dynamic and is to be given as a part of the system configuration.

Documents are grouped into corpuses, such as Wikipedia and CommomCrawl. The system supports multiple corpuses simultaneously and provides an easy way how to switch between them.

Search results are presented as snippets, which are parts of the text of the documents that match the given query. Every snippet can be repeatedly extended until the full document length is reached. Alternatively, it is also possible to display the whole document in a dialog window. Users can also navigate to the original source, from which the semantically enhanced document was created. The metadata should be presented as tooltips.

Part of the solution is a user management system, which handles different types of users with different privileges. Each user can save his search configuration, including the amount of results per page. For each corpus, they can also select and save only a subset of the metadata it provides.

User roles

In Enticing, users can have several roles. The roles are ordered and each of the more privileged roles have some extra privileges on top of the previous ones.

- User – can only edit his own settings and select metadata for each corpus
- Corpus maintainer – can add, edit and delete corpus configurations
- Admin – can manage user roles of other users

The text above can be summarized using the use case diagram shown in the figure 4.1. It provides use cases from the specification above as well as a few other that were added later during development.

4.2 EQL – Enticing Query Language

EQL is a language which can be used to query semantically enhanced documents on the Enticing platform. The queries can be as simple as a few words, but also very complex, containing logical operators, subqueries over multiple indexes or constraints further limiting the results.

The rest of the section is structured as follows. First part is a practical guide that shows how to use EQL. It starts with a simple query and gradually adds more operators to it to satisfy the requirements. Then, a list of all operators that EQL supports is given along with their description.

Practical guide

Let's start with a simple query, whose purpose will be to search for documents talking about Bonaparte visiting Jaffa[1]. Most people would probably write a query like the following one.

```
Bonaparte visits Jaffa
```

For sure, this query is a good starting point, but there is quite a lot of place for improvement. First thing that one might be tempted to do is to relax the ordering of the words, so that sentences like *Jaffa visited by Bonaparte* are matched as well. But since this requirement is quite common, it is actually the default behavior. On the other hand, the ordering of words can be enforced using the **order** operator. For some queries this might be very useful. Let's consider the following one.

```
Gauguin < influenced < Picasso
```

This query will search for any document where these three words appear in the specified order. Sometimes it might be important for the words to stand next to each other. For that, the **sequence** operator can be used.

```
"Gauguin influenced Picasso"
```

[1]It is a reference to a painting called Bonaparte Visiting the Plague Victims of Jaffa, see https://en.wikipedia.org/wiki/Bonaparte_Visiting_the_Plague_Victims_of_Jaffa for more details.

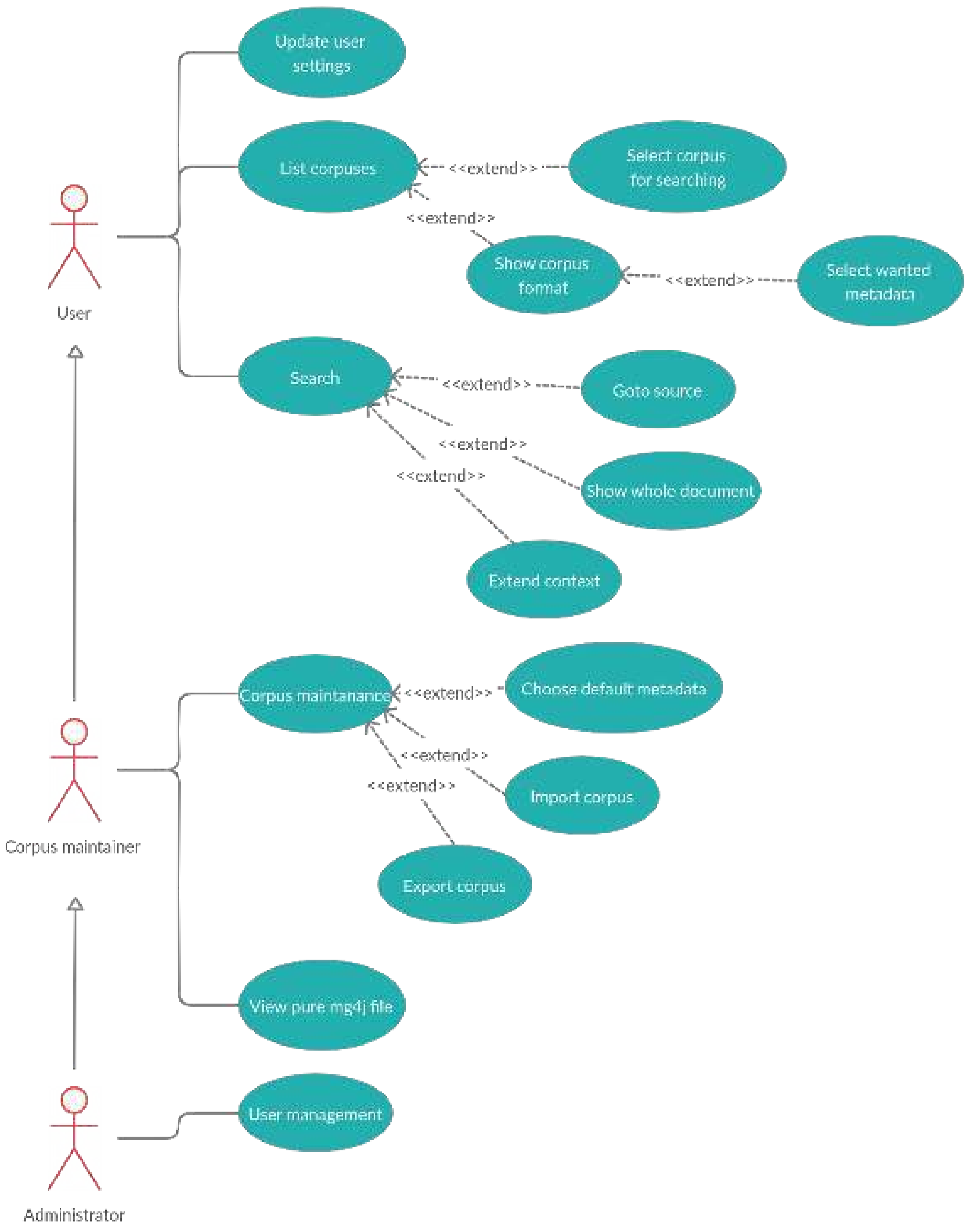


Figure 4.1: **Use case diagram**

Let's go back to the original *Bonaparte visits Jaffa* now. Even though the order is not important, it might be important for these words to appear close to each other. Without any modification, the words can be anywhere within the document. To change that, **Context constraints** can be used.

```
Bonaparte visits Jaffa ctx:par
Bonaparte visits Jaffa ctx:sent
```

These two queries require the words to appear in one paragraph or in one sentence, respectively. Let's go with the latter option for now. Using just one verb might be a bit too specific. Let's add a second option, the verb explore. The **or** operator can be used for that.

```
Bonaparte ( visits | explores ) Jaffa ctx:sent
```

Now the search would return documents containing words Bonaparte, Jaffa and either visited or explored, or both. Note that the parenthesis are not necessary in this case. The query `Bonaparte visits | explores Jaffa ctx:sent` has the same meaning, but the first version might be a bit more explicit. Without the parenthesis, it might look like both words on each side are part of the `or`, which is not the case. To change the precedence, parenthesis can be used: `(Bonaparte visits) | (explores Jaffa) ctx:sent`.

Let's focus on the verbs again. There might be documents talking about Bonaparte visiting Jaffa, but the shape of the verb *visit* can be different. For example it might be written in the past tense. To match documents like that, **index** operator can be used.

So far the query used only the default index, which is *token.* This index contains words from the original document. The metadata about words can be found in the other indexes. For example, index *lemma* contains the lemma for each word.

```
Bonaparte lemma:visit|explore Jaffa ctx:sent
```

So far only basic indexes, logic operator and constraints were used. But the semantic enhancement offers a lot more. In EQL, the query can contain entities instead of exact words. For example, there might be a sentence talking about Bonaparte and Jaffa, but it uses pronouns instead of the direct names. Using entities in the query, such sentences can be matched as well.

```
person.name:Bonaparte lemma:visit|explore Jaffa ctx:sent
```

Word `Bonaparte` was replaced with entity `person.name:Bonaparte`. In this example *person* and *name* are just another indexes that can be queried, just like *lemma* or *token.* This query will be looking for an entity of type person whose name is Bonaparte. It is also possible to look for any person using the **index** operator.

```
nertag:person lemma:visit|explore Jaffa ctx:sent
```

Nertag is an index that contains the type of the entity. Jaffa can be changed the same way as Bonaparte.

```
person.name:Bonaparte lemma:visit|explore  place.name:Jaffa ctx:sent
```

By applying a few operators, the resulting query became more generic and therefore it matches more documents, while keeping the same requirements.

There is still one more group of operators to talk about, **global constraints**. To illustrate what they are for, let's search for two artists and a relationship between them. The skeleton of the query is `artist influenced artist`, but a few operators are needed to make the query more generic.

```
nertag:person|artist < lemma:((influence|impact) | (pay < tribute))
< nertag:person|artist ctx:par
```

This query is looking for documents where there is an entity followed by a word or words followed by an entity. Both entities should be of type person or artist. The text between them must contain either a single word, whose *lemma* is either influence or impact, or 'pay tribute', again in any form thanks to using the lemma index. On top of that, this whole query is limited to a single paragraph. But unfortunately, there is a problem. There is no requirement that says that these two entities should be different. To express such requirements, **global constraints** can be used. Firstly, parts of the query that should be used within the constraint are identified using the **assignment operator**, then the constraint itself can be expressed.

```
influencer:=nertag:person|artist <
lemma:(influence|impact) | (pay < tribute)  <
influencee:=nertag:person|artist ctx:par
&& influencer != influencee
```

The constraint above ensures that the two artist are different. Notice that no attributes were specified for the comparison. The default index for comparison is *nerid*, which contains a unique identifier for each entity. It is possible to specify any attribute for comparison.

```
    ... && influencer.name != influencee.name
```

In the end, one more example is given.

```
a:=nertag:person|artist < lemma:visit < b:=nertag:person|artist
place.name:Barcelona nertag:event^event.date:[1/1/1960..12/12/2012]
ctx:par && a != b
```

This query is looking for documents talking about one artist visiting another artist in Barcelona during an event that happened between 1/1/1960 and 12/12/2012. The context is limited to a single paragraph and global constraints are used to ensure that the two artists are different. Notice that when specifying an event, `nertag:event ^ event.date` was used. `nertag:event` requires that entity of type event has to be in the document while `event.date` adds a specific requirement on that event. This form is actually required internally while processing the query, but it is **not** necessary to do it explicitly, as the compiler performs such rewrites on its own. For example `place.name:Barcelona` is compiled to `nertag:place ^ place.name:Barcelona`. It was mentioned here to provide an opportunity to introduce the **alignment** operator `^`, which allows to express multiple requirements over the same word or entity in the document.

Operators

The operators can be divided into four categories.

Basic operators

- **Implicit and** – `A B` – If no operator is specified, **and** is chosen implicitly. That means that all mentioned words have to be in the document, in any order.
- **Order** – `A < B` – A should appear before B, but they do **not** have to be next to each other.
- **Sequence** – `"A B C"` – A, B and C have to appear in this order next to each other.
- **And** – `A & B` – Both A and B have to be in the document, in any order.
- **Or** – `A | B` – At least one of A, B has to be in the document.
- **Not** – `!B` – B should not be in the document.
- **Parenthesis** – `(A | B) & C` – Parenthesis can be used to build more complex logic expressions.
- **Proximity** – `A B ~ 5` – A and B should appear at most 5 positions apart.

Index operators

To work with meta information, one has to specify index for querying. That can be done using the following operators.

- **Index** – `index:A`

Look for document, where value A is present at given index. For example, `lemma:work` will match any document in which any word, whose lemma is work, appears, e.g. works, working, worked, etc.

It is also possible to ask more complex queries such as `index:A|B|C`. Apart from another index operator or entity operator, a query of any form can be used.

When working with entities, it is possible to query their attributes as well. Querying all the people with name Picasso can be done using query `person.name:Picasso`.

If the index contains only integers or dates, **range operator** can be used to specify an interval, such as `date:[1/1/1970..2/2/2000]` or `person.age:[20..30]`.

- **Align** – `index1:A ^ index2:B`

The align operator allows to express multiple requirements over one word. For example, it is possible to look for a noun, whose lemma is do. This query can be written as `pos:noun ^ lemma:do`.

Context constraints

The default context for searching is the whole document. For more granular queries, it is possible to add the following limitations to the query using a special index context or ctx, both options work.

- **Paragraph** – `ctx:par` – Limits the query to one paragraph only.
- **Sentence** – `ctx:sent` – Limits the query to one sentence only.

Document constraints

It is also possible to limit the search to a single document or a group of them. This can be done using a special entity called document or doc. Both options work.

- **UUID** – `doc.uuid='...'` – Limits the query to the document with specified UUID.

UUID identifies the document internally, but it is not possible to determine it by looking at the original document. The user knows it only after seeing the document meta information, so he can't type it into his first query. To compensate for that, the *title* of the document or its *url* can be used as well. But please note that these are not always unique, therefore results from multiple documents matching the requirements might be returned.

Global constraints

Sometimes it is necessary to specify a relationship between multiple entities that can't be expressed using the previous operators. One example might be searching for documents talking about two artists influencing each other. A simple query for that would be the following.

```
nertag:artist < lemma:influence < nertag:artist
```

But there is a problem with this query. It might return irrelevant snippets, because there is no requirement that the two artist should be different. This is where the global constraints come into play. The global constraint is a predicate which is separated from the query by the symbols `&&`. The constraint consists of one or more equalities and inequalities connected using logical operators `and, or, not` and parenthesis, if necessary.

In order to use the global constraints, relevant parts of the query have to be identified first.

- **Assignment** – `x:=A` – Assign an identifier to a certain part of the query.

Afterwards, it is possible to write queries with global constraints.

```
1:=nertag:artist < lemma:influence < 2:=nertag:artist && 1.nerid != 2.nerid
```

To increase the readability, string identifiers can be used instead of numbers.

```
influencer:=nertag:artist < lemma:influence < influencee:=nertag:artist
&& influencer.nerid != influencee.nerid
```

4.3 Component architecture

This section contains the high-level architecture of Enticing. It consists of 4 main components. The diagram in the figure 4.2 presents these components along with their relationships. Each of them will now be described.

Webserver

Webserver is the first and only component common users will interact with. It is also the only one that is meant to be publicly accessible. All requests should pass through it before being forwarded further into the system. It exposes an API that can be used by any

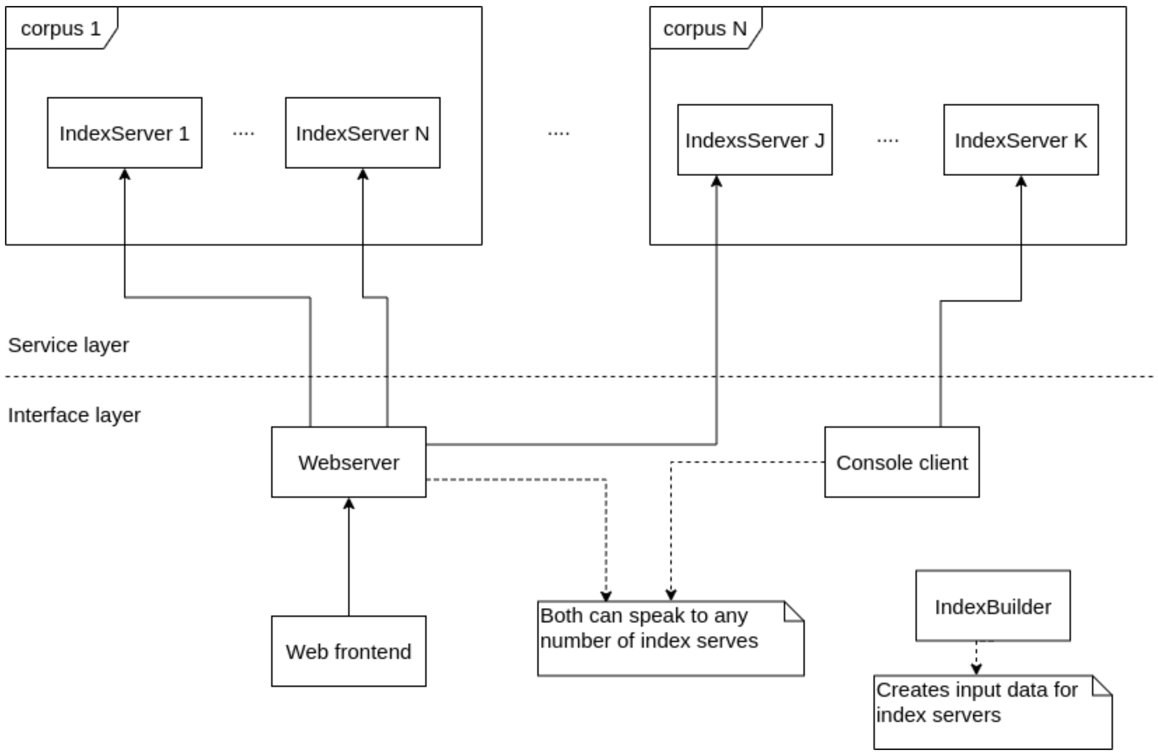


Figure 4.2: **Components** Components of the system

third-party service to submit a search query. It is bundled with a single page JavaScript application that serves as a Graphical User Interface (GUI) of the system.

Apart from being the entry point for queries, Webserver's responsibilities also include user management and search settings management.

ConsoleClient

Sometimes, especially for research purposes or testing, it is useful to submit a query from the command line. ConsoleClient was designed for this use case. It allows to submit a list of queries in a batch and collect all the results into a file. It also has an interactive shell. Queries can be sent to the Webserver, a group of IndexServers or just a single IndexServer. ConsoleClient also supports making performance measurements.

IndexServer

IndexServer is a component that maintains a set of indexed documents and exposes an API for querying them. This API should be accessible only internally, from the Webserver or the ConsoleClient. It is not meant to be directly reachable from outside the system.

Internally, the set of documents is divided into collections. Each of these collections is handled concurrently.

This component is meant to be deployed multiple times on multiple machines to handle bigger text corpuses.

IndexBuilder

This component is a command line tool responsible for preprocessing documents and creating indexes that are later used by IndexServers. The process of indexing is both time and resource consuming, that's why it is handled separately and not directly from the IndexServers.

4.4 Module architecture

Components described in the previous section consist of different modules, which can be classified into two groups. *Library modules* implement functionality shared across multiple components. *Executable modules* contain component specific logic. They group together necessary *library modules* and produce jar archives that can be executed on the JVM. This section describes the architecture of these modules. The dependencies between them are visualized in the diagram in the figure 4.3.

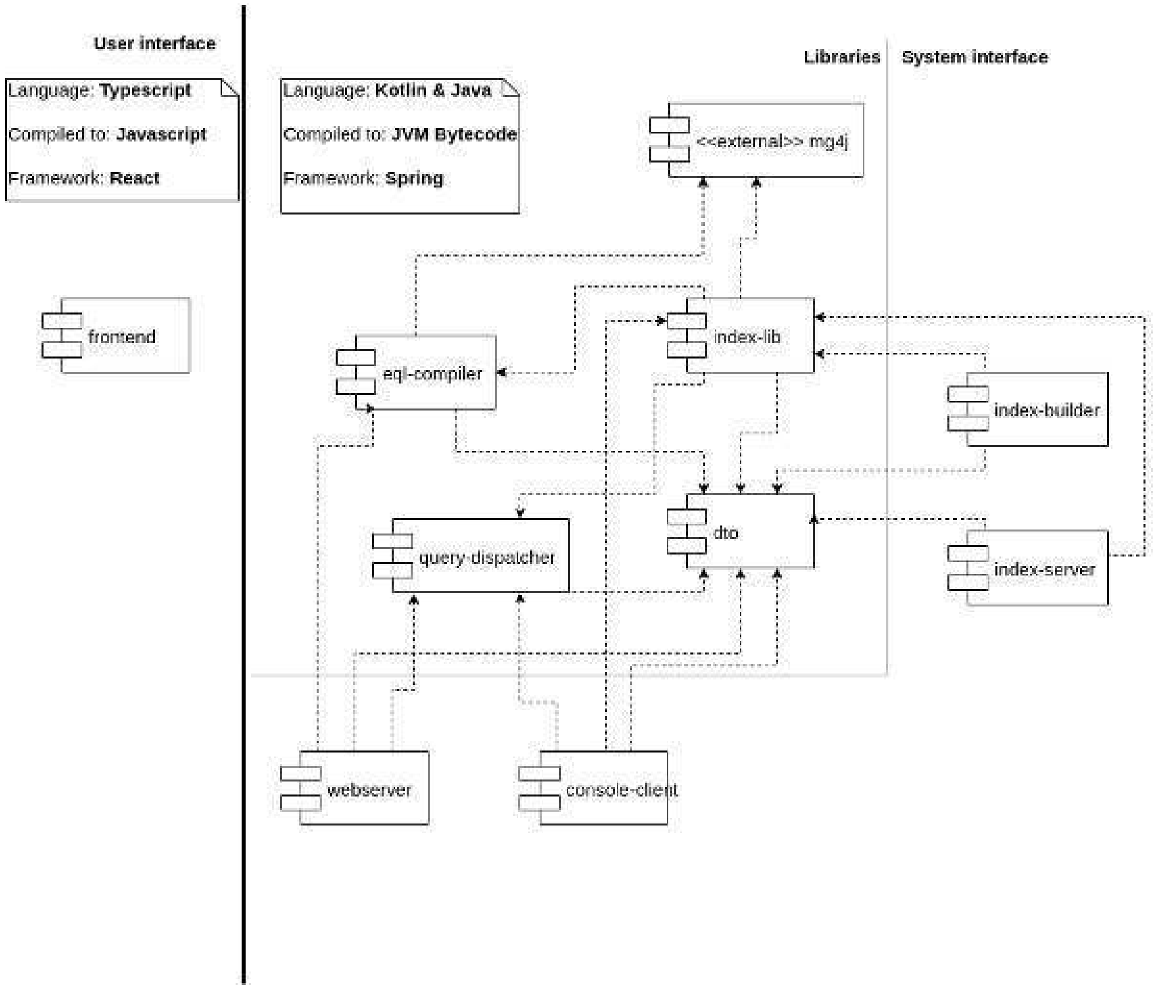


Figure 4.3: **Modules of the system**

Library modules

First, individual library modules will be presented.

DTO

This is the central library module. It contains the domain logic shared by all components. This includes but is not limited to the following.

- Data transfer objects (DTOs) to pass data between components.
- Functions transforming DTOs on each layer of the system.
- Configuration classes and related functions.
- Custom logging micro-library with support for remote logging.
- API classes for each component, which one component can use to communicate with another.

WebClient

WebClient is a JavaScript single page application that provides a graphical user interface over the API of the Webserver. It can be used to submit queries, manage search settings and view search results. The flow between screens in the WebClient is described using the screen diagram in the figure 4.4. It is bundled with the Webserver component, which serves it on the root URL.

IndexLib

IndexLib is a module whose responsibility is to perform indexing and searching. It is built on top of MG4J, which internally performs most of the operations. IndexLib provides an MG4J-agnostic interface. The rest of the system should not be dependent on MG4J, so that different search engines can be integrated easily. This module also contains search algorithms used for matching documents using EQL. These algorithms already use the MG4J-agnostic format, so that they can be used to evaluate results from different engines in the future.

EqlCompiler

As the name suggests, EqlCompiler analyses and compiles EQL queries into the query language used by MG4J. The process of compilation an EQL query is described in the communication diagram in the figure 4.5. First, the text query is parsed using a generated parser. Then the resulting parse tree is transformed into EQL Abstract Syntax Tree (AST). Then the semantic analysis is performed. Equivalent MG4J query can then be generated by traversing the AST. To integrate a different search engine, all that has to be done is to provide an algorithm transforming the AST into the format used by the new engine.

QueryDispatcher

QueryDispatcher contains the implementation of an algorithm, which dispatches queries to a given set of nodes in an iterative way until the wanted amount of snippets is collected or

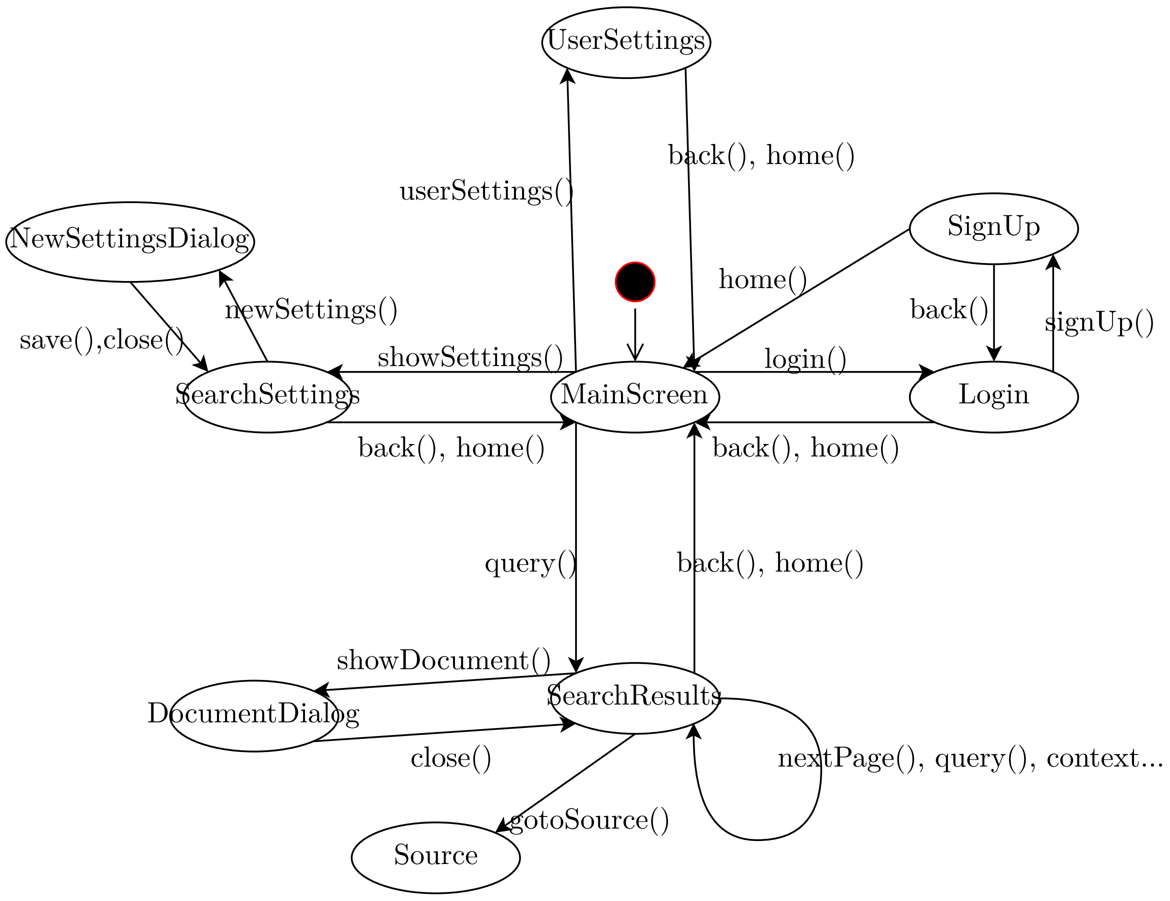


Figure 4.4: **WebClient screen diagram**

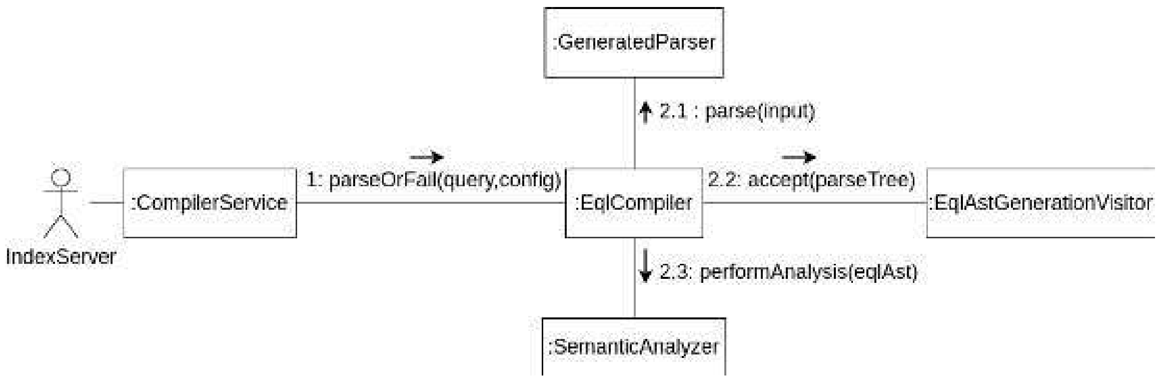


Figure 4.5: **EQL query compilation**

no nodes can provide more. In each round, all remaining nodes are queried in parallel. The pseudocode for the algorithm is the following.

```
def queryDispatcher(query,initialNodes,snippetCount):
    snippets = []
    nodes = initialNodes
    while snippets.size < snippetCount and nodes.isNotEmpty():
       results = await parallel_call(query,nodes)
       nodes = []
       for result in results:
            if result.isSuccess and result.snippets.isNotEmpty():
                 snippets.addAll(result.snippets)
                 if result.offset:
                    nodes.add(NodeWithOffset(result.node,result.offset))
    return snippets
```

It is possible to prove that this algorithm always terminates by exploring the conditions of the while loop. It will evaluate to true if and only if the accumulated amount of snippets is less than the wanted amount and there are still some nodes to query. The list of nodes to query is always cleared inside the loop. New nodes are added to it only if they successfully provided some results. Therefore the amount of snippets in each iteration either increases or there will be no nodes for the next iteration. It can therefore be concluded that the algorithm always terminates, because it either collects enough results or has no more nodes to process.

This algorithm was intentionally designed to be generic with regards to how the nodes are queried. Inside Enticing, there are two places where it is used. The first place is in the Webserver or ConsoleClient, when dispatching queries to IndexServers. The second place is in the IndexServer, when dispatching to individual collections. The flow of a single query is visualized in the figure 4.6.

Executable modules

The description of the Webserver, IndexServer, ConsoleClient and IndexBuilder as components was covered in 4.3. The corresponding *executable modules* contain the component specific business logic. As the name suggest, these components can be executed. Each of them contains an entry point which starts the corresponding component.

4.5 Transferring annotated text

This section covers data structures used within Enticing to transfer annotated text of semantically enhanced documents. Two different *result formats* are supported and both of them can use 4 different *text formats* to transfer the actual text.

Result format

There are two supported result formats – *snippet* and *identifier list*. The meaning of snippet has already been described in the chapter 2. Snippet is generally a very useful format, but not for every use case. Sometimes all that is needed are specific patterns within the snippet. *Identifier list* was designed for this use case. It contains only the fragments that were matched by given EQL identifiers.

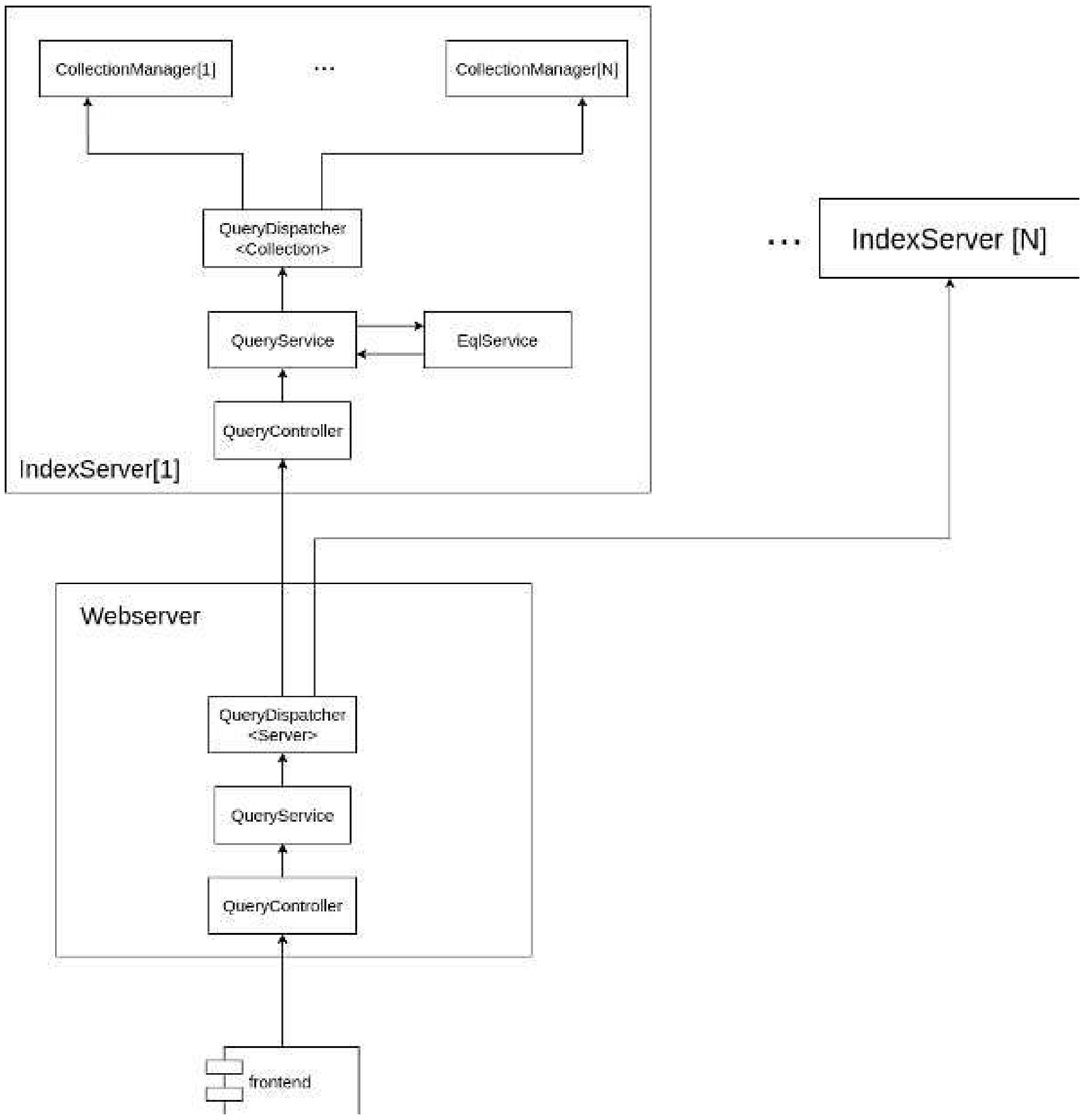


Figure 4.6: **QueryFlow** How a query flows through Enticing

Text format

Enticing supports four text formats, each of them is useful for different use case.

Plain text

Plain text is the simplest format. As the name suggest, it contains only the plain text from the document, without any metadata. Sometimes it is really all that is needed. It can be useful for the ConsoleClient, since showing metadata in the terminal is rather complicated. It is also useful format for testing. Another advantage is that it is the most compact, so it can come in handy when a lot of text has to be transferred over a slow network.

HTML

This format follows the structure of HTML files. Each word that has metadata is wrapped within a *<span>* tag and the metadata are encoded as attributes with prefix *eql*. The matched elements are wrapped within *<b>* tags. Its advantage is that there are many tools supporting HTML files, so it should be fairly easy to read its content. Its main disadvantage is that it is the hardest to parse. Its properties make it a suitable format when the content should have metadata (otherwise plain text is sufficient), but it should also be displayed directly as a text file, without any specialized graphical user interface. An example is given below.

```
<b>
    <span eql-word eql-position="10" eql-tag="NN"
    eql-lemma="job" eql-parpos="12">
        job
    </span>
</b>
<span eql-word eql-position="11" eql-tag="VVZ"
eql-lemma="require" eql-parpos="10">
    requires
</span>
```

String with annotations

In this format, the annotations are encoded as intervals over the plain text. The format has four parts. Its structure will be presented using a simple example.

```
{
    "text": "job requires expertise in..."
    "annotations": {
        "w-0": {
            "content": {
                "position": "10",
                "tag": "NN",
                "lemma": "job",
                "parpos": "12"
            }
        }
    },
    "positions": [{
        "annotationId": "w-0",
        "match": {"from": 0, "size": 3},
        "subAnnotations": []
    }],
    "queryMapping": [{"textIndex": {"from": 0, "size": 3},
                        "queryIndex": {"from": 0, "size": 1}}]
}
```

Text represents the actual text of the snippet. *Annotations* is a map of annotations, each of them has a unique identifier. *Positions* describe the locations of these annotations. The definition of an annotation and its usage are separated to avoid unnecessary duplication. This way, each annotation can be used multiple times. *QueryMapping* describes how the query matched the document. This format is not very suitable for reading directly, but it is much easier for automated parsing. Unfortunately, it is rather hard to generate. It requires a lot of computation and object allocations. On top of that, the description by intervals also has another bad property. When the text changes, all intervals starting from the point of change all the way to the end of the snippet have to be updated. And this is exactly what happens when the context is extended, so it is quite a common use case. Also, the rendering pipeline in the frontend demands a different format, so post-processing in the browser is necessary and it can be quite expensive. Exact measurements were not taken, but they didn't even have to be. When multiple results arrived to the frontend, the slowdown was so significant, that it was unacceptable. Therefore the *text unit list* format was designed as a replacement. Since the format generating pipeline was already implemented, *String with annotations* is still supported, but it is not used by the frontend anymore.

Text unit list

As mentioned in the previous subsection, this format was added later as a replacement for *String with annotations.* It covers the same use case – transferring annotated text for client apps to visualize – but compared to the previous one, it is even a bit easier to read it directly. However, that is only a fortunate side effect. Let's describe its main features. Its structure is visualized in the diagram in the figure 4.7. The text is a list of TextUnits, which can be either a Word, an Entity or a QueryMatch. Entity can contain multiple words, QueryMatch can contain multiple TextUnits. The advantages of the format are the following. It is very close to the format used for analyzing documents in the IndexLib, therefore its creation is a straightforward process. Changes in its structure are always local – they can be performed without the need to update the rest of the document. And on top of that, this format is suitable for the frontend without any transformations. It can be rendered directly as it is. Its disadvantage is that it duplicates the entity information. However, the same technique as in the previous format can be applied if the memory overhead becomes a problem. An example is given below.

```
{
    "content": [
        {
            "type":"queryMatch",
            "subunits": [
                {
                    "type": "word",
                    "content": ["10", "job", "NN", "job", "12"]
                }
            ]
        }, {
            "type": "word",
            "content": ["11", "requires", "VVZ", "require", "10"]
        }
    ]
}
```

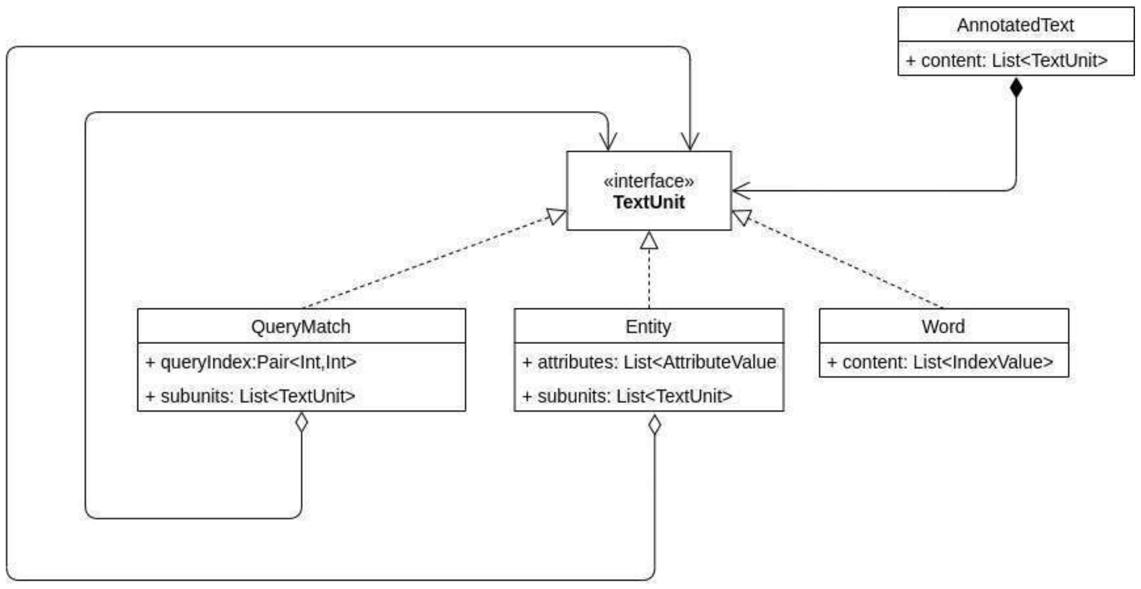

AnnotatedText
+ content: List<TextUnit>
«interface»
TextUnit
QueryMatch
+ queryIndex:Pair<Int,Int>
+ subunits: List<TextUnit>
Entity
+ attributes: List<AttributeValue
+ subunits: List<TextUnit>
Word
+ content: List<IndexValue>

Figure 4.7: **Text unit list** class diagram

Chapter 5

Used technology

This chapter describes the programming languages, frameworks and libraries used when building Enticing. Even though there are many components and modules in the project, they can be divided into two groups, based on the environment in which they will be deployed. Frontend technology is discussed in 5.1, backend in 5.2.

5.1 Frontend

Frontend components are supposed to run in a web browser. In particular, the WebClient module falls within this category.

The mainstream language for developing web applications is JavaScript. However, its dynamic nature makes it hard to develop complex systems and even for a simple one, a static type system can be a great benefit [10]. Therefore, TypeScript[1] was used instead.

According to the previous chapter, WebClient is a single page JavaScript web application. Nowadays, there are many frameworks that help with building such applications. Among those, React[2] was chosen for its simplicity, performance and, last but not least, the developer experience.

React is focused purely on the user interface. Therefore, another library called Redux[3] was added for the state management. This combination is quite common nowadays, because it provides a clean separation between UI and business logic and also it forces the developer to think about the state of the app and its transitions [1].

TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Its goal is not to replace the language, only to add a static type system to it. It is a proper superset, meaning that any JavaScript program is a valid TypeScript program. The type system was designed in such away to allow for gradual adoption. The typechecks are optional and all warnings can be ignored if necessary. This allows developers to incrementally add types to existing projects instead of doing it all at once, which would quite often not be feasible [10].

[1]https://www.typescriptlang.org/
[2]https://reactjs.org/
[3]https://redux.js.org/

React

React is a JavaScript library for building user interfaces (UI) that is developed and maintained by Facebook. It is declarative and component based. In the philosophy of React, the view is a pure function of the state. Developers therefore only declare how the view should be rendered based on the state and every time the state changes, the library triggers the minimal necessary amount of re-rendering. The basic building blocks in React are components, encapsulated reusable pieces of the user interface that manage their own state or depend on the state of their parents. Complex user interfaces are created by composing these components together [17].

Redux

Redux is a state container for JavaScript applications. Its idea is that the state should be centralized, normalized and, most importantly, never changed directly. Instead, it is protected inside a container, which allows read-only access and change detection. When any component wants to change the state, it does so by dispatching an action. The actions are received by reducers, pure functions that can produce a new state based on the action and the old state [1].

CodeMirror

CodeMirror is a versatile text editor implemented in JavaScript for the browser [9]. It is specialized for editing code and provides support for a lot of mainstream languages out of the box. Fortunately for us, it also supports creating custom language modes. It is used inside the WebClient for the „smart“ search bar. To integrate it with React, the library React Codemirror[4] was used.

5.2 Backend

Since the backend components should interact with MG4J, using a JVM-based language is a necessity. Among those, Kotlin[5] was chosen. It is fully interoperable with Java and it provides many useful features such as data classes, extension methods, null safety and, last but not least, full support for functional programming [6].

For developing web services, Spring Framework[6] was used. It is the most popular framework for building backend applications on the JVM [11].

For parsing the custom query language, Antlr[7] parser generator was used.

Kotlin

Kotlin is a multiparadigm multiplatform programming language developed by JetBrains. It supports both object oriented and functional programming and allows developers to mix them to get the best from both worlds. Thanks to the fact that it can be compiled into JVM bytecode, JavaScript and LLVM bitcode, it can be run on almost any platform. It originally started as a replacement for Java. On one hand, JetBrains wanted more expressive language

[4] https://github.com/JedWatson/react-codemirror
[5] https://kotlinlang.org/
[6] https://spring.io/
[7] https://www.antlr.org/

having modern features such as data classes, extension methods, functions as types, etc. On the other hand, they already had a big codebase written in Java, which they could not abandon. Therefore they decided to create a new language which would have the features they wanted, but which would be fully interoperable with Java, so that they could reuse their existing libraries and tools [6].

Spring Framework

Spring originally started as a dependency injection framework and gradually became an umbrella project consisting of a huge amount of subprojects handling different needs of enterprise developers [11]. Inside Enticing, the following subprojects are used.

- Spring MVC – For creating REST interfaces.
- Spring Data – For connecting to the database.
- Spring Boot – For creating a standalone Spring based application with minimum amount of configuration.

Antlr

Antlr is a parser generator created by Terence Parr. It allows developers to define the syntax of the language using a very expressive grammar, supporting iterations, left recursion and even semantic predicates. The grammar is then compiled into a parser. Apart from the parser itself, a visitor and a listener are created as well to allow developers to easily iterate over the parse tree and transform it into custom data structures based on their needs [15].

Chapter 6

Implementation of the search engine

This chapter focuses on the implementation of the search engine Enticing. The chapter 4 described the architecture of its core component and modules. This chapter extends it and provides more technical details. It is divided into self contained sections, each of them describing one part of the platform or one interesting piece of functionality that deserves a deeper explanation. The section 6.1 discusses the implementation of eager result loading. The topic of search result pagination and offsets is covered in 6.2. The section 6.3 dives deeper into EQL and explains how the compiler and searching was implemented. The process of encapsulation MG4J is described in 6.4. The design of the web user interface is given in 6.5. Part of that interface is a smart search bar, whose implementation is discussed in 6.6. Finally, Enticing configuration DSL is introduced in 6.7.

6.1 Eager result loading

For the testing scenario with only a few IndexServers, the QueryDispatcher algorithm described in the previous chapter was working well. However, once that number increased[1], an unfortunate property of the algorithm became apparent. It waits for all servers to reply before returning any result. The servers, on which the platform is deployed, are used for other tasks as well, quite often computational intensive ones. And in the current settings, one busy server causes all the results to be delayed, which is of course not acceptable. Therefore the original implementation was extended to allow for eager result loading.

It was done in the following way. The algorithm now accepts a callback, which is triggered every time new results arrive. This is the only change necessary in the QueryDispatcher and a rather simple one. The real difficulty is in sending the results to the client. Two approaches were considered – *pushing* and *pulling*, each of them having different implications on the resulting API.

In the *pushing* scenario, results are sent to the client as messages. This is of course the cleanest solution, but it requires a channel from the server to the client through which the data should be sent. It can be achieved using WebSockets[2], which support full-duplex communication between the server and the client. Unfortunately, WebSockets were not included in the original design of the Webserver as their usage did not seem necessary.

[1]Currently, about 50 servers are used.

[2]https://cs.wikipedia.org/wiki/WebSocket

Including them now would require significant effort on both client and server side and a lot of code would have to be updated. That's why the solution was discarded as not feasible. However, it is still possible to implement it later on as an extension.

In the *pulling* scenario, the client submits a query and then keeps asking for new results in a loop, until everything is delivered. The client code is quite simple, but it requires a storage for these temporary results on the server. It would be bad to put them in a database, because they are ephemeral and a database is for persistent data. In the end, a simple custom in-memory cache was used. This cache is storing the data in a ConcurrentHashMap, which is a Java thread safe map implementation. It has to be thread safe, because it will be used from multiple threads. The callback of the QueryDispatcher stores the data to the cache and the calls from the client remove them. Two important follow-up questions had to be answered.

The first one is how to identify and remove old entries that were not retrieved. It can happen for example because the user closes the page in the middle of submitting a query. If no cleanup is performed, the map will keep growing until the program runs out of memory. Therefore a timestamp was added to each entry and old entries are periodically removed.

The second one is what should be the keys for the cache. It has to be something that will always be unique for a given user and it can't be his login, because queries can be sent by anonymous users as well. The first considered approach was generating a UUID on the server and sending it back to the client immediately. But that would change the old API, which waited for all results. That was unwanted, the old approach is perfectly fine for the ConsoleClient use case. Eager loading should be optional, not a default. There is no need to trigger it for clients who do not need it. Therefore it was decided to push this option downstream. If the client wants to use eager loading, it generates the unique id, includes it in the query and then uses it when pulling results. The WebClient implementation generates random UUIDs, so the collision chances are acceptably low.

6.2 Pagination and offsets

In a distributed environment with unpredictable delays, even a simple looking task like pagination actually presents a challenge. Thanks to the distributed nature of the searching in Enticing, the results from each IndexServer will likely come in different order every time. And inside IndexServers, the same problem appears with results from collections. It is of course possible to sort the results, but that does not play well with eager result loading presented above. The results should be delivered as soon as possible and sorting them in the GUI would result in an unpleasant glitch as the already visible results would have to be shifted from time to time. Therefore, it was decided to accept the non-determinism rather then to fight with it. As a result of that, the pagination information cannot be encoded in the URL with deterministic outcomes. Even though the same results appear each time the page is loaded, their order is likely to be different. This poses additional burden on the user, because if he wants to share his results with someone else, he has to write a query that is sophisticated enough to return only a few snippets. Document restrictions were originally designed for this use case, but they turned out to be useful even outside of it.

Even the structure of the offset is actually quite complex. Normally, one integer is sufficient. A careful reader might object that result size is also important and he is right. However, the size is decided by the user of the system, therefore it does not have to be encoded in the reply. The user knows it and can submit it with the next query. In our scenario, even the simple offset into one collection actually has two parts – *document offset* and *result offset*. The first one indicates which document to start from, while the second indicates what should be the first returned snippet from the document. This offset is merged with offsets from other collections, which yields a map-like structure for the IndexServer offset. Then, on the webserver, each of these IndexServer offsets has to be merged again. The resulting structure is then a map of a map of the collection offsets. Fortunately, it is possible to completely hide this complex structure from the user. The API of the Websever has two endpoints for querying. The first one is for submitting the query and the second one is for requesting more results. The offset is stored in the HTTP session of the user. The second endpoint does not need any parameters, because it can load everything from the session.

6.3 Enticing Query Language

This section covers the implementation details of the EQL compiler.

Syntax and Semantic analysis

Almost every compiler consists of the following parts [2].

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Optimizations (sometimes optional)
- Target code generation

EQL compiler is no exception to that rule. As mentioned earlier, Antlr is used for the lexical and syntax analysis. However, the parse tree which Antlr returns after parsing is too low-level for more advanced analysis and transformations that are done in the backend of the compiler. Therefore a custom Abstract Syntax Tree was designed.

Taking inspiration from the Antlr itself, visitors and listeners were designed and implemented to allow each step in the pipeline to easily traverse and modify the AST. The difference between a visitor and a listener is that the visitor has to explicitly call itself on the children of the current node to proceed. On the contrary, the listener consists of callbacks that are automatically executed for each node in the AST. Both of these traversals are useful, each for a different type of operation. Transformation from Antlr parse tree to EQL AST is done using *EqlAstGeneratingVisitor*.

The next step in the pipeline is semantic analysis. Since there are many checks that should be done and their number can increase over time, a flexible solution was necessary. Here, the inspiration was taken from IntelliJ IDEA[3], an open source IDE for Java and Kotlin developed by JetBrains. Each semantic check is a subclass of *EqlAstCheck*. These subclasses are forced to implement a method which takes as input a node in the AST, symbol table and a few other useful objects. *EqlAstCheck* is generic and using its type variable, subclasses can specify what type of AST node they should be run from. The semantic analyzer takes a list of these checks as input, groups them by the type of node and then traverses the AST of the query and runs appropriate checks for each node.

[3]https://github.com/JetBrains/intellij-community

Unfortunately, some of these checks depend on the execution of other checks. For example, there is a check which verifies that all indexes and entities in the query are from the corpus configuration. Then there are two checks for document restriction and context restriction, which remove these pseudo nodes from the AST and save the information about restrictions in the root node. If the validation check runs before the transformation checks, it reports unknown index *context* and unknown entity *document*, since they are not part of the corpus configuration. Similar dependencies appear for other combinations of checks. To cope with that, the checks were grouped into phases. Each check, which is dependent on the execution of another, is put into a later phase. After running each phase, the semantic analyser looks for errors and proceeds only if none are found. This also allows for early exit, since more sophisticated analysis is not necessary if simple semantic errors are found.

After the semantic analysis, the AST is ready to be used. The compiler instance, which is integrated into the webserver, stops here, only returning errors that were encountered back to the frontend. In the IndexServer, the AST is used in two places. The first one is generating MG4J query, which is done using *Mgj4QueryGeneratingVisitor*. The second one is in the postprocessing of the results, which is the topic of the following subsection.

EQL-based Searching Algorithms

Unfortunately, MG4J does not return a very precise description of how the query matched the document. It only returns the overall intervals for each index. This is enough for creating basic snippets, but not enough to highlight the parts of the snippets that were matched by the query. Also, EQL has a concept of identifiers, which should be highlighted as well, but MG4J has no support for them. Therefore, after retrieving a document from the SearchEngine, it is necessary to perform a post-processing to compute which parts of the documents were matched by which nodes in the AST. This information is precise enough to provide the highlighting, however the computation is time and space consuming. It essentially requires to re-implement the searching functionality of MG4J, which is not an easy task. On the other hand, it gives a chance to create more powerful semantics for the query, which can be more complex than the underlying technology.

MG4J uses Minimal-Interval Semantics, which significantly reduces the number of returned snippets and which allows them to use very efficient algorithms [5]. However, it has drawbacks. Imagine a query about a place and a person. Now lets say that there is a document containing three different people and three places and the intervals between them overlap. MG4J would return only one interval connecting one person with one place, but in Enticing, it is preferred to return all three combinations that occur, even though they overlap. MG4J does not support that, but it will at least return the document and then the post-processing generating all possible combinations can be performed. The con of this solution is that the number of results can grow exponentially. To cope with that, it was decided to limit the number of snippets per document.

6.4 Encapsulating MG4J

Even though MG4J is a useful and powerful library, a different indexing technology might be chosen in the future. Therefore it is important to encapsulate MG4J as tightly as possible and provide MG4J agnostic abstractions around it. This boundary consists of a set of interfaces and data transfer objects that can be found in the package *c.v.f.k.e.index.boundary*. The core class of the IndexLib module, *CollectionManager*, which interacts with the indexing library, only depends on the boundary classes from the package above. Therefore the process of integrating new indexing library would consist only from implementing the interfaces from the package above and initializing the *CollectionManager* with them.

The process of searching using a *CollectionManager* is visualized in the figure 6.1. First, search query and offset are given to the *CollectionManager*. It delegates the request to the *SearchEngine*, which is an interface providing methods for querying the collection. Documents are represented using an interface *IndexedDocument*. Results from the *SearchEngine* are then given to the *PostProcessor*, which analyses the document using EQL searching algorithms and finds all interval matching the query. Previous two steps might be executed in a loop until wanted amount of results is accumulated or there are no more document matching the query. In the end, results are genereated using *ResultCreator*, which is responsible for transforming the document and the query-match information into the result format specified by the query.

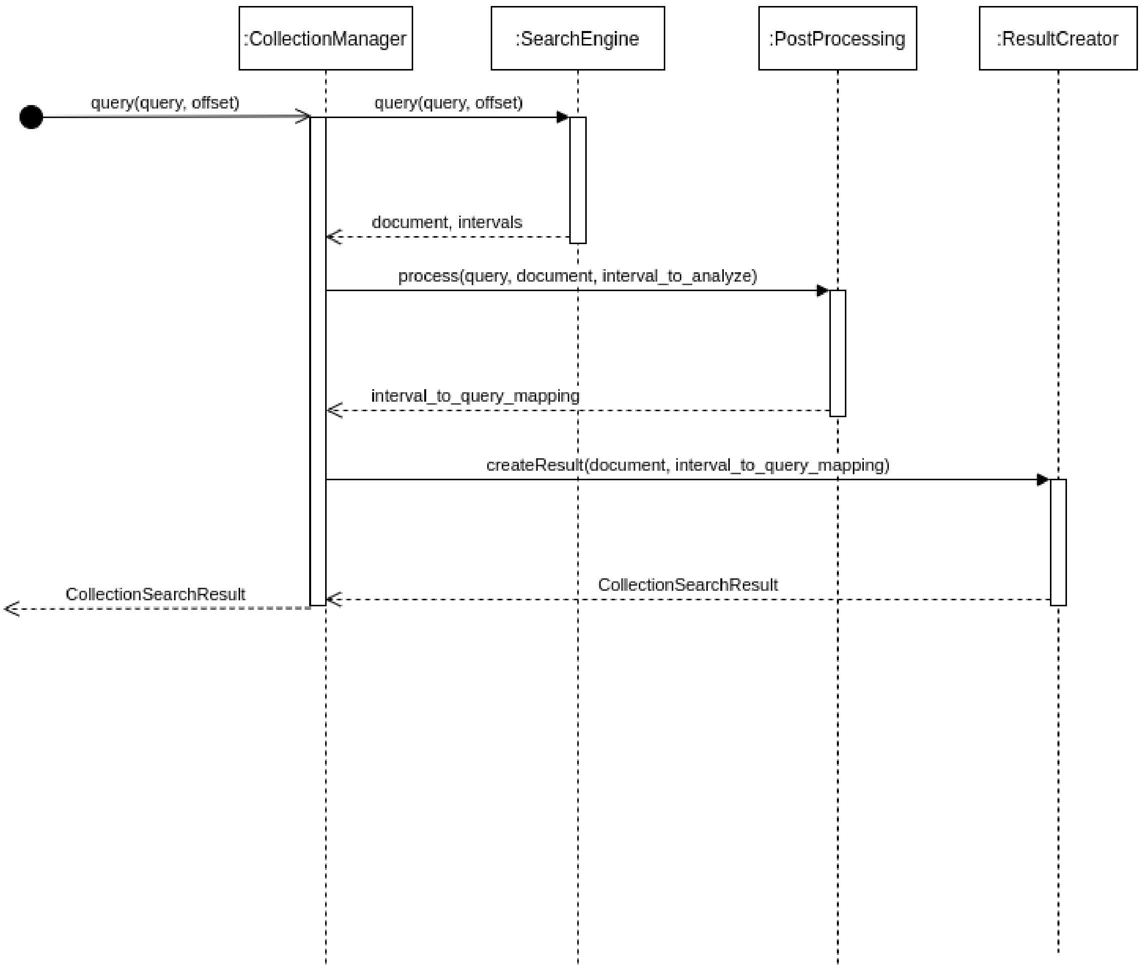
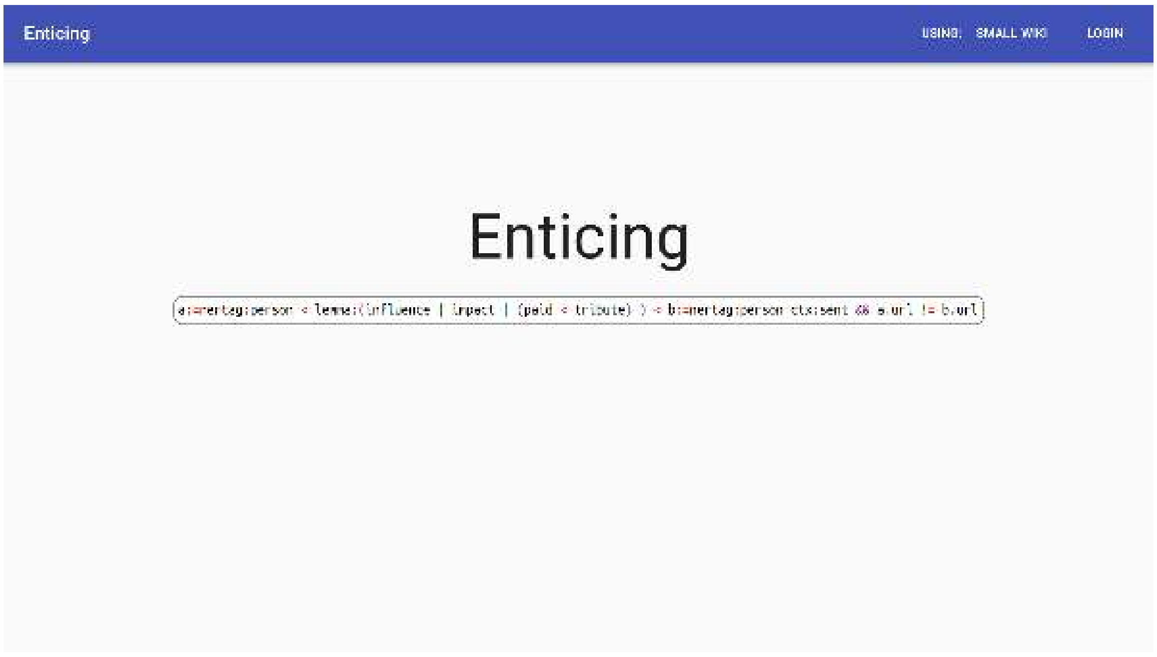


Figure 6.1: **Collection Manager** Searching in a collection

6.5 User interface

Good user experience is a must these days. In this section, screenshots of the web frontend are provided and the thought process behind the design is discussed.

The first screen user sees is the main page, which you can see in the figure 6.2. It was inspired by the main page of Google[4]. It should be minimalistic, containing only the bare minimum one needs to start searching, which is a search bar of course. One difference between Google and Enticing is that in Enticing users can specify corpuses which should be searched. It was decided to put this option into the main toolbar at the top of the screen, so that is does not pollute the structure of the page, while still being easily accessible.



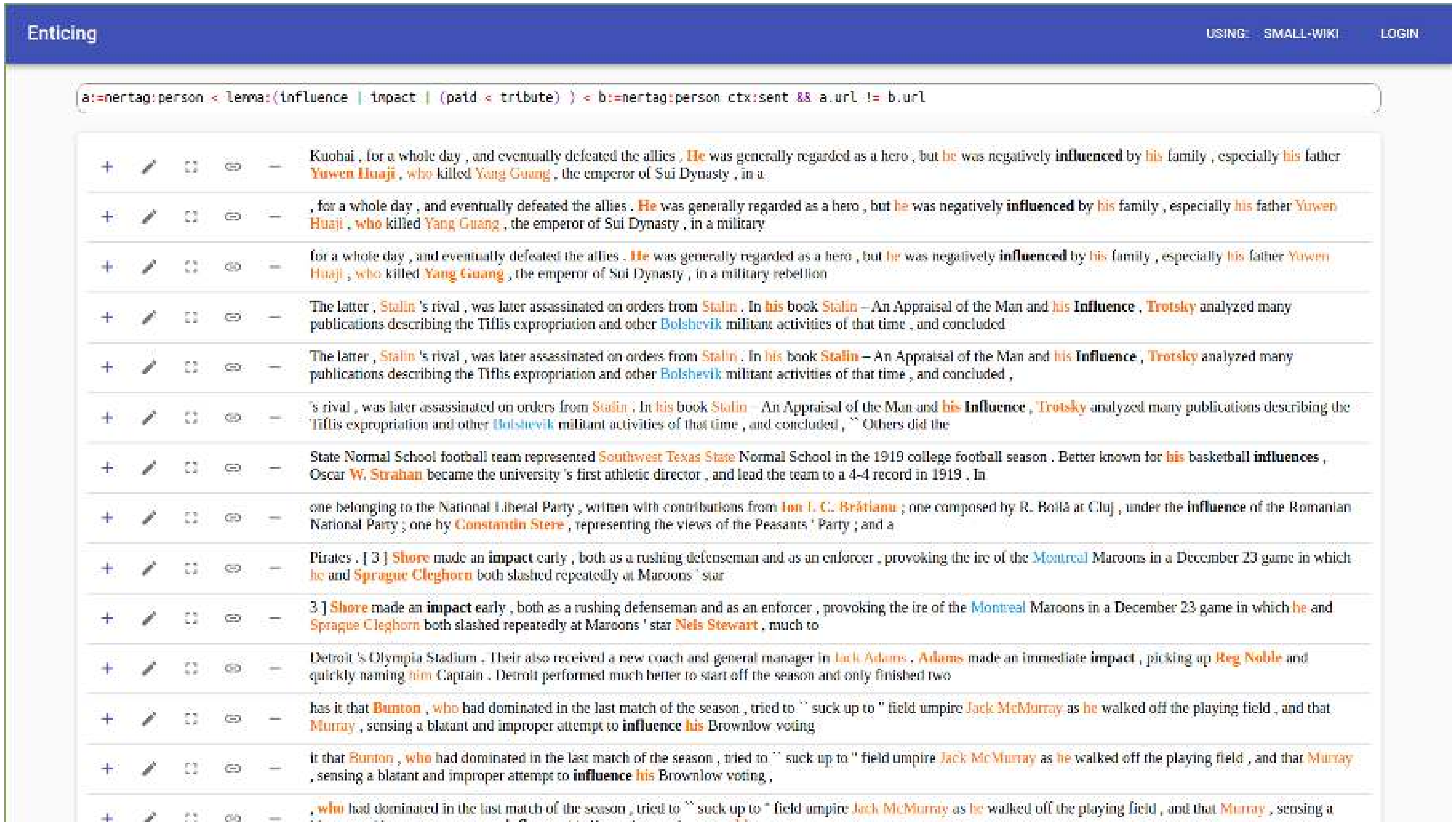
```
a:=nertag:person < lemma:(influence | impact | (paid < tribute) ) < b:=nertag:person ctx:sent && a.url != b.url
```

Figure 6.2: **Main page**

After a search query is submitted, the user is redirected to the search result page, which you can see in the figure 6.3. This page uses the same toolbar as the main page. The search bar is positioned right below it to allow users to submit another query quickly if necessary. Search results presented as snippets follow. On the left of each snippet are buttons, which can be used to extend its context, open full document, go to the original source or reduce the search scope only to given document. Each of these buttons has an explanatory tooltip, which pops up when user hovers over it. On the right is the content of the snippet.

The biggest challenge when presenting the results is displaying the metadata. All words have metadata assigned to them and on top of that, some sequences of words represent entities with attributes. It was decided to display the entities using different colors. User can configure which color to use for which type of entity in the settings. For displaying all metadata, tooltips are used. These tooltips appear when user hovers over a word in the snippet. An example of a tooltip with simple word can be seen in the figure 6.4 and an example of an entity can be seen in the figure 6.5. A careful reader can notice that these

[4] https://www.google.com/



```
a:=nertag:person < lemma:(influence | impact | (paid < tribute) ) < b:=nertag:person ctx:sent && a.url != b.url
```

Kuohai , for a whole day , and eventually defeated the allies . **He** was generally regarded as a hero , but he was negatively **influenced** by his family , especially his father **Yuwen Huaji** , who killed Yang Guang , the emperor of Sui Dynasty , in a

, for a whole day , and eventually defeated the allies . **He** was generally regarded as a hero , but he was negatively **influenced** by his family , especially his father Yuwen Huaji , **who** killed Yang Guang , the emperor of Sui Dynasty , in a military

for a whole day , and eventually defeated the allies . **He** was generally regarded as a hero , but he was negatively **influenced** by his family , especially his father Yuwen Huaji , who killed **Yang Guang** , the emperor of Sui Dynasty , in a military rebellion

The latter , Stalin 's rival , was later assassinated on orders from Stalin . In **his** book Stalin – An Appraisal of the Man and his **Influence** , **Trotsky** analyzed many publications describing the Tiflis expropriation and other Bolshevik militant activities of that time , and concluded

The latter , Stalin 's rival , was later assassinated on orders from Stalin . In his book **Stalin** – An Appraisal of the Man and his **Influence** , **Trotsky** analyzed many publications describing the Tiflis expropriation and other Bolshevik militant activities of that time , and concluded ,

's rival , was later assassinated on orders from Stalin . In his book Stalin – An Appraisal of the Man and **his** **Influence** , **Trotsky** analyzed many publications describing the Tiflis expropriation and other Bolshevik militant activities of that time , and concluded , `` Others did the

State Normal School football team represented Southwest Texas State Normal School in the 1919 college football season . Better known for **his** basketball **influences** , Oscar **W. Strahan** became the university 's first athletic director , and lead the team to a 4-4 record in 1919 . In

one belonging to the National Liberal Party , written with contributions from **Ion I. C. Brătianu** ; one composed by R. Boilă at Cluj , under the **influence** of the Romanian National Party ; one by **Constantin Stere** , representing the views of the Peasants ' Party ; and a

Pirates . [3] **Shore** made an **impact** early , both as a rushing defenseman and as an enforcer , provoking the ire of the Montreal Maroons in a December 23 game in which he and **Sprague Cleghorn** both slashed repeatedly at Maroons ' star

3] **Shore** made an **impact** early , both as a rushing defenseman and as an enforcer , provoking the ire of the Montreal Maroons in a December 23 game in which he and Sprague Cleghorn both slashed repeatedly at Maroons ' star **Nels Stewart** , much to

Detroit 's Olympia Stadium . Their also received a new coach and general manager in Jack Adams . **Adams** made an immediate **impact** , picking up **Reg Noble** and quickly naming him Captain . Detroit performed much better to start off the season and only finished two

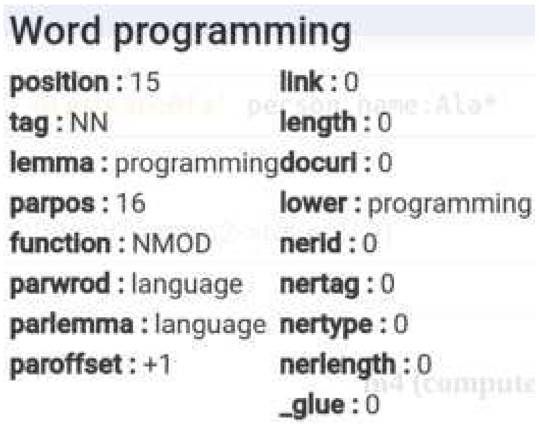
has it that **Bunton** , who had dominated in the last match of the season , tried to `` suck up to '' field umpire Jack McMurray as he walked off the playing field , and that Murray , sensing a blatant and improper attempt to **influence** **his** Brownlow voting

it that Bunton , **who** had dominated in the last match of the season , tried to `` suck up to '' field umpire Jack McMurray as he walked off the playing field , and that Murray , sensing a blatant and improper attempt to **influence** **his** Brownlow voting ,

, **who** had dominated in the last match of the season , tried to `` suck up to '' field umpire Jack McMurray as he walked off the playing field , and that Murray , sensing a

Figure 6.3: **Search result page**

pictures are a bit transparent. This was done on purpose. These tooltips can take quite a lot of space and this way the user still sees what is located under them.



Word programming

position : 15	**link** : 0
tag : NN	**length** : 0
lemma : programming	**docurl** : 0
parpos : 16	**lower** : programming
function : NMOD	**nerid** : 0
parwrod : language	**nertag** : 0
parlemma : language	**nertype** : 0
paroffset : +1	**nerlength** : 0
	_glue : 0

Figure 6.4: **Tooltip with a simple word**

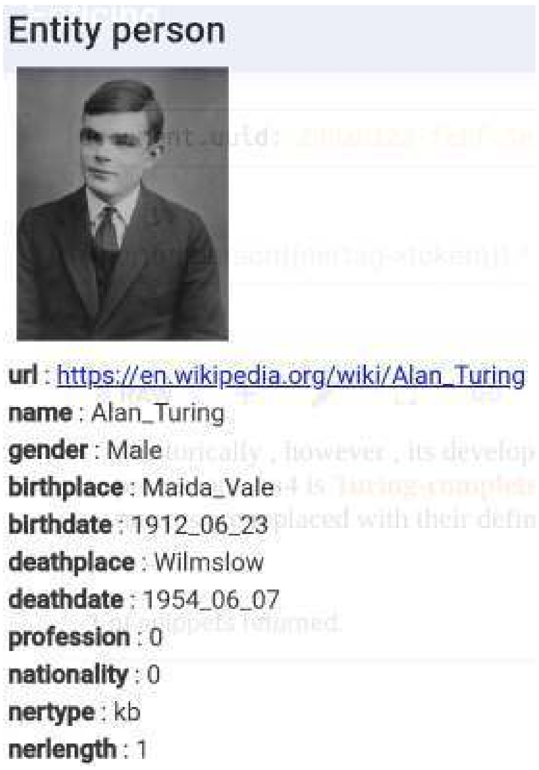
6.6 Smart search bar

Since EQL is a formal language with well-defined structure and semantics, giving the user just a plain HTML input field seemed unsatisfactory. The feedback should be given while typing the query. Syntax highlighting would be a good start, full syntax and semantic analysis could follow. Autocomplete would also be great. The flow should seem similar to writing a program using an IDE.

Unfortunately, there are not so many options for smart text editors in the browser. And the ones that exist usually have some bad properties. They are either not for programming or they support only syntax highlighting and nothing more or they are very old and hard to use. The remaining editors actually do what is required, but they are way too complex for a simple search bar.

After a while, it was decided to take inspiration at JetBrains again, particularly at the way Kotlin Playgroud[5] is implemented. Kotlin Playground is a minimalistic online IDE in which users can experiment with Kotlin in their browser. Fortunately, it is open source[6], so the answers could be found by reading the source code. They are using an editor called CodeMirror[7]. The first impression categorized it as option three from the list above – old and hard to use. However, after a deeper analysis, it was fortunately discovered that only the first part is true. Using it is actually quite easy. Syntax highlighting is only a matter of providing regular expressions describing tokens. Syntax and semantic analysis were a bigger challenge. Re-implementing the whole EQL compiler in JavaScript seemed wrong, since it would be a lot of work and then two compilers would have to be maintained instead of one. The inspiration was taken from the Kotlin Playground again, where they were sending the code to the backend for analysis. This options allowed to use the existing Kotlin implementation of the compiler and just expose it as an endpoint on the Webserver. It of course introduces some delay, but the experience showed that it was acceptable. This

[5]https://play.kotlinlang.org/
[6]https://github.com/JetBrains/kotlin-playground
[7]https://codemirror.net/


Entity person

url : https://en.wikipedia.org/wiki/Alan_Turing
name : Alan_Turing
gender : Male
birthplace : Maida_Vale
birthdate : 1912_06_23
deathplace : Wilmslow
deathdate : 1954_06_07
profession : 0
nationality : 0
nertype : kb
nerlength : 1

Figure 6.5: **Tooltip with an entity**

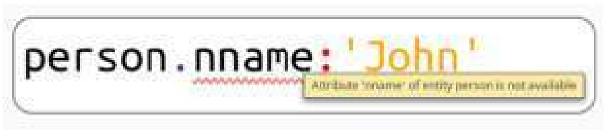

```
person.nname:'John'
```
Attribute 'nname' of entity person is not available

Figure 6.6: **Search bar displaying an error message**

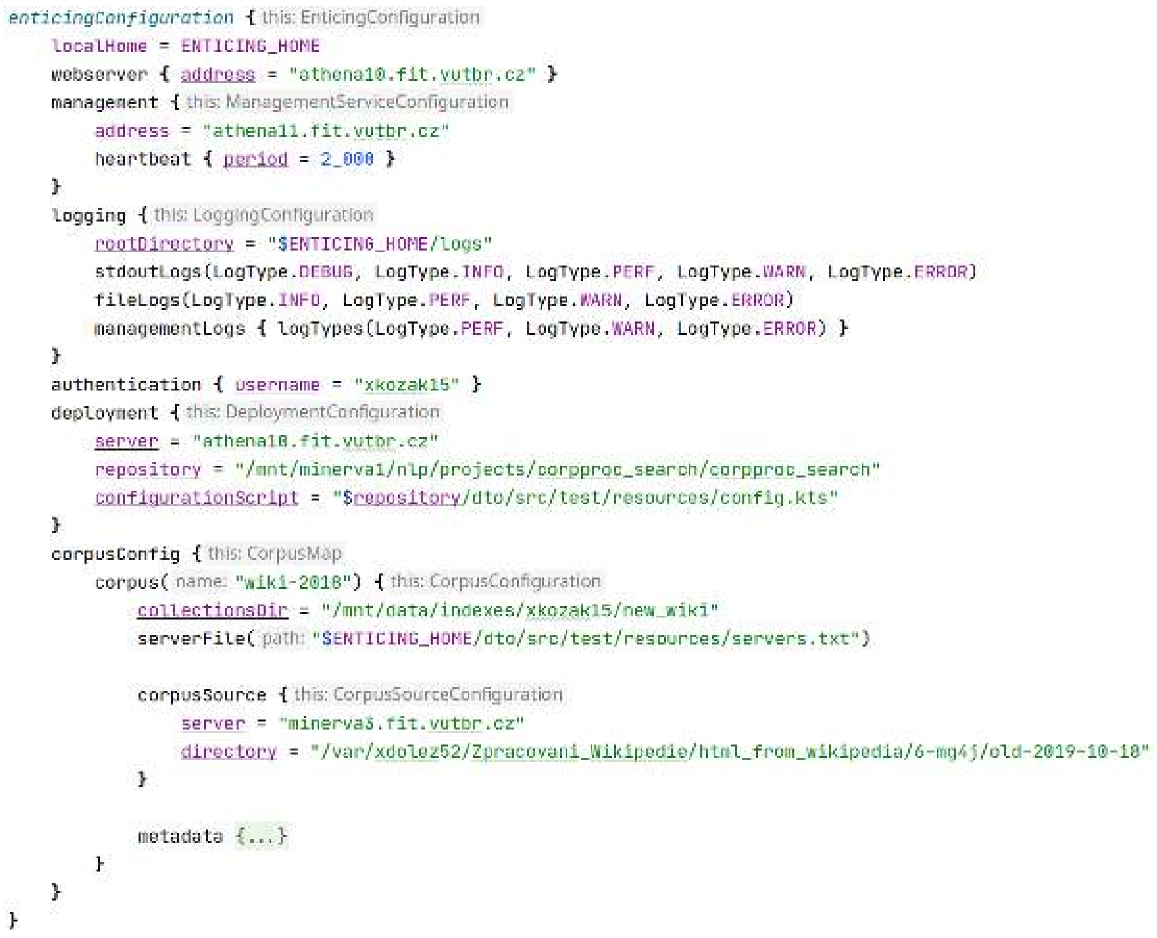
way it was managed to get syntax and semantic analysis reasonably fast. Example of the searchbar displaying an error message can be seen in the figure 6.6. Unfortunately, autocomplete was not implemented yet. CodeMirror supports it, so it can be a nice and challenging follow-up work.

6.7 Enticing Configuration DSL

Kotlin has features which allow developers to create their own internal Domain Specific Languages (DSLs). The advantage of using them is that they provide a higher level of abstraction compared to pure Kotlin, which makes the resulting code easier to read and therefore it improves the maintainability. Even though some of these abstractions can be suitable even for general programming, one of the use cases, in which they really shine, are configuration DSLs. These DSLs provide a more powerful alternative to XML or JSON based configurations. Any piece of Kotlin code can be used within them and the IDEs can provide validation and autocomplete for them.

A custom DSL for Enticing was created which allows one to easily configure the whole platform in a way that should be readable even for a non-programmer. Example of a configuration written using the DSL can be seen in the figure 6.7. Please note the syntax highlighting and additional semantic information, which the IDE presents to the user for free.

After the DSL was ready, the question became how to use it. Configuration is something rather dynamic, so including it directly into the source code did not seem like a good option. Fortunately, Kotlin has a scripting API, which handles this use case very well. Using this API, a file containing the configuration can be loaded by each component at runtime and used to configure it.



```
enticingConfiguration { this: EnticingConfiguration
    localHome = ENTICING_HOME
    webserver { address = "athena10.fit.vutbr.cz" }
    management { this: ManagementServiceConfiguration
        address = "athena11.fit.vutbr.cz"
        heartbeat { period = 2_000 }
    }
    logging { this: LoggingConfiguration
        rootDirectory = "$ENTICING_HOME/logs"
        stdoutLogs(LogType.DEBUG, LogType.INFO, LogType.PERF, LogType.WARN, LogType.ERROR)
        fileLogs(LogType.INFO, LogType.PERF, LogType.WARN, LogType.ERROR)
        managementLogs { logTypes(LogType.PERF, LogType.WARN, LogType.ERROR) }
    }
    authentication { username = "xkozak15" }
    deployment { this: DeploymentConfiguration
        server = "athena10.fit.vutbr.cz"
        repository = "/mnt/minerva1/nlp/projects/corpproc_search/corpproc_search"
        configurationScript = "$repository/dto/src/test/resources/config.kts"
    }
    corpusConfig { this: CorpusMap
        corpus( name: "wiki-2018") { this: CorpusConfiguration
            collectionsDir = "/mnt/data/indexes/xkozak15/new_wiki"
            serverFile( path: "$ENTICING_HOME/dto/src/test/resources/servers.txt")

            corpusSource { this: CorpusSourceConfiguration
                server = "minerva3.fit.vutbr.cz"
                directory = "/var/xdolez52/Zpracovani_Wikipedie/html_from_wikipedia/6-mg4j/old-2019-10-18"
            }

            metadata {...}
        }
    }
}
```

Figure 6.7: **Example of Enticing Configuration DSL**

Chapter 7

Testing and evaluation of the search engine

This chapter covers methods used for testing and evaluation of the Enticing platform. First, general overview of the types of tests that were used is given in 7.1. The section 7.2 describes the testing module, which contains the integration and performance tests of the whole platform. The section 7.3 focuses on the tests of the EQL searching algorithms. Results of the performance measurements are given in 7.4. In the end, the description of the Continuous Integration and Continuous Delivery setup is presented in 7.5.

7.1 Types of tests

Since the project is very complex, multiple different types of tests were used together to ensure its quality. Each module has its own *unit tests* verifying its functionality. When unit testing individual units of code, it is sometimes necessary to mock the behavior of their dependencies. For that, *mocking libraries* were used. Different modules are combined together into components and *integration tests* verify the communication between them. The searching algorithms were mostly verified using *functional tests*, ensuring that correct outputs are returned for specified inputs. However, writing these functional tests is time consuming, because for each tested query, one has to find suitable test documents and then specify what should be matched. Therefore the number of these tests is limited. To increase the confidence with the quality of the searching algorithms, a declarative DSL was created. It can be run on any group of mg4j files, veryfying that the returned results have declared properties. More details about these tests will be given in the section 7.3. *Performance tests* were implemented to measure the performance of the engine. They were very useful when optimizing the search performance.

7.2 Overview of the testing module

The *integration* and *performance tests* were created in a special testing module. New management module is used to start, monitor and kill components under test. For querying the components, ConsoleClient module is used. This way the tests are done by using real API calls, therefore the components are tested in a very realistic scenarios. However, these tests are very expensive, therefore they are not included in the CI pipeline.

The first group of tests in the module are *integration tests.* Using them, the following properties are verified.

- After submitting a query, results are successfully returned.
- The structure of the results is correct.
- The results match the requirements specified in the query.
- The same results are returned every time the same query is submitted.
- Pagination works as expected. Making one big query or adequate amount of small ones should produce the same results.

The tests are generic with respect to the querying mechanism. This way, the same suite can be executed on three layers – IndexServer, QueryDispatcher and Webserver.

The second group are *performance tests.* In these, the queries are submitted multiple times and the time of their execution is measured. It is then possible to compute the average value, the deviance, min, max, etc. These tests can be performed on the three different layers mentioned above.

7.3 Testing the searching algorithms

One of the biggest challenges when testing the platform was testing the searching algorithms based on EQL. They form the very core of the search engine functionality and therefore their correctness is of the highest importance. Since the interactions in the search engine are very tightly coupled, it was decided to create *functional tests* verifying that correct outputs are produced for given inputs. The tests can be categorized into two groups.

Tests performed on dummy documents

In these tests, the documents are handcrafted to contain specified keywords and then they are send to the search engine. Since the documents were created manually, it is exactly known what results should be returned, therefore it is easy to check them. However, the setup phase of these tests is very time consuming, therefore their amount is limited.

DSL-based tests performed on real data

Another option is to use real mg4j files for the tests. The advantage of this approach is that the inputs are real documents, therefore the tests are more realistic. The disadvantage is that it is necessary to read the documents beforehand to know what kind of results should be expected.

To automate this process, another DSL was created. It allows one to declare what properties should hold for every result returned for a given query. A small test engine was then implemented, which queries given set of documents and and then verifies that the results given by the *CollectionManager* are valid. This of course does not guarantee that all results are returned, but it at least guarantees that the returned results are meaningful. These tests can also be scaled very easily simply by adding more mg4j files as input. Example of one test written using this DSL is given below.

```
@DisplayName("That Motion three")
@Test
fun simpleQuery() = forEachMatch("That Motion three") {
    forEachInterval("all three words should be there") {
        val text = textAt("token", interval)
        verify("that" in text) { "'that' should be in '$text'" }
        verify("motion" in text) { "'motion' should be in '$text'" }
        verify("three" in text) { "'three' should be in '$text'" }
        verifyLeafCount(3)
    }
}
```

7.4 Performance measurements

Apart from the correctness itself, performance is very important. Users expect search results to be delivered fast and therefore the whole system has to be optimized to satisfy that. In order to perform any optimizations, measurements have to be taken first, to make sure that the optimizations work as expected.

The first measurements were taken in the end of March 2020. Their results are presented in the table 7.1. The duration of querying one IndexServer was measured, as it is the most important use case to optimize. Wanted amount of snippets was 20. Tested IndexServer instance was deployed on KNOT server *knot01.fit.vutbr.cz*. This server has Intel Xeon E5-2630 2.3 GHz processor with 15MB cache and 6 cores. Total ram size is 65536 MB. The IndexServer instance was handling 10 mg4j files, which together had 2.9 GB and contained 24371 documents. A list of queries was created and then submitted 100 times, taking the average, deviation, min and max value. The testing started with simple single word queries, which were then combined together using logical operators. In the end, context restrictions and global constraints were added. Contrary to what was expected, more complex queries were not always significantly slower. The number of occurences of searched terms and their locations played a significant role as well. For example, there were only 1285 matches of the word water, but there were more than 10000 entities of type person. Therefore less documents had to be iterated when providing 20 results. You can also notice a significant slowdown in the query *water nertag:person nertag:location ctx:sent.* The author initially thought that it was because the context restrictions operators were not very optimized, but the second measurements showed otherwise.

The second measurements were taken in the middle of May 2020. You can see their results in the table 7.2. The same server and the same set of documents were used. However, there were a lot minor updates of the searching algorithms in the meantime, which targeted both bugs and performance issues. The context restriction evaluation was updated as well. As you can see, the searching became faster, but the difference is unfortunately not as big as the author hoped, especially for the query *water nertag:person nertag:location ctx:sent.* It seems that even though new context restriction evaluation had a positive impact, a lot of documents have to be processed to provide snippets, which slows down the search.

It is also important to note that other processes are running on the tested server, which influences the results. On top of that, the whole end to end time for each query is measured, which is also a subject to a lot of noise from the environment. On the other hand, this way of testing is as close to what the user experiences as possible. However, it might be

EQL Query	Average[ms]	Deviation[ms]	min[ms]	max[ms]
water	711.4	65843.12	337	1823
nertag:person	155.4	2263.76	104	421
nertag:location	177.73	24321.48	93	835
water nertag:person	503.0	35445.24	355	1259
water nertag:person ctx:sent	423.15	8891.11	321	771
nertag:person nertag:location	130.24	1919.32	96	433
nertag:person nertag:location ctx:sent	287.09	2490.72	220	463
water nertag:person nertag:location	453.82	14603.47	331	1056
water nertag:person nertag:location ctx:sent	4317.23	16767.16	4071	4828
nertag:person nertag:person	119.28	1071.40	96	403
nertag:person nertag:person ctx:sent	124.48	828.09	100	330
a:=nertag:person b:=nertag:person	117.31	699.074	97	287
a:=nertag:person b:=nertag:person ctx:sent	130.61	667.64	104	247
a:=nertag:person b:=nertag:person && a.url != b.url	126.33	801.48	98	263
a:=nertag:person b:=nertag:person ctx:sent && a.url != b.url	353.91	10034.46	250	801

Table 7.1: Results of the first performance measurements

EQL Query	Average[ms]	Deviation[ms]	min[ms]	max[ms]
water	191.41	196.86	118	2033
nertag:person	35.46	52.47	22	546
nertag:location	33.43	30.50	23	313
water nertag:person	303.10	122.36	220	1071
water nertag:person ctx:sent	149.35	108.73	93	846
nertag:person nertag:location	41.10	37.48	24	298
nertag:person nertag:location ctx:sent	154.03	42.00	116	349
water nertag:person nertag:location	787.29	105.52	657	1030
water nertag:person nertag:location ctx:sent	4016.33	147.78	3860	5075
nertag:person nertag:person	30.72	25.89	23	286
nertag:person nertag:person ctx:sent	42.69	20.67	31	163
a:=nertag:person b:=nertag:person	33.81	15.44	24	113
a:=nertag:person b:=nertag:person ctx:sent	44.62	19.85	31	220
a:=nertag:person b:=nertag:person && a.url != b.url	34.73	17.22	24	138
a:=nertag:person b:=nertag:person ctx:sent && a.url != b.url	306.94	74.51	254	776

Table 7.2: Results of the second performance measurements

beneficial to create a suite of performance tests targeting just the searching algorithms in isolation in the future.

Also, please note that the system is still being developed, so the results presented here might not be aligned with the performance you are experiencing when using Enticing. Hopefully, the system will be even faster by the time you experiment with it.

7.5 Continuous Integration and Continuous Delivery

Since running the tests became more and more time consuming after a while, a Continous Integration (CI) was configured for the GitHub repository of the project. After every commit to the master and release branches, the full test suite is run remotely. The CI was setup using CirlceCI[1].

[1] https://circleci.com/

Chapter 8

Deployment of the search engine

This chapter covers the deployment of the platform. In a system with more than 50 servers involved, deploying, monitoring and logging efficiently becomes a real challenge. The section 8.1 covers the deployment scripts that were used in the beginning. Custom logging library that was developed as a part of this project is introduced in 8.2. Finally, the section 8.3 introduces the management module that was added as an extension of the platform. Once the platform was successfully deployed, it was presented at Excel@FIT 2020[1], where it received an Expert Panel Award [13].

8.1 Deployment scripts

The first chosen approach for deployment were command line scripts. The high level functions orchestrating the execution were written in Python 3, the low level interactions were written in Bash. The main script loaded its configuration from *.ini* file and then executed actions specified as command line flags.

This approach was working well for a while, mainly because the number of running components was rather small. However, once that number increased, the drawbacks of this solution became more and more visible. It gave no feedback about the current state of the platform. If something had gone wrong, it would be good to be notified about that. And even better, automatic error recovery should be used whenever possible.

Another problem was testing. These scripts started expensive computations on a large amount of servers. Therefore it was important to test them properly. For the Python scripts, *unit tests* were written, but they could not be run from the CI pipeline. The developer had to start them manually every time and it was of course easy (and sometimes tempting) to forget about that. And testing the shell scripts presented even bigger challenge.

The *.ini* file was also a bit problematic. It essentially contained the same information as given in the Enticing configuration script. So some unfortunate redundancy was involved. It would be much better to load the configuration from the well-tested statically typed Kotlin DSL instead of duplicating it in this file. But loading Kotlin code into Python script is of course quite complicated.

For the reasons stated above, the decision was made to create a more robust deployment solution, preferably in Kotlin, so that the already implemented business logic could be reused whenever possible.

[1]http://excel.fit.vutbr.cz/

8.2 Logging

It is important to gather all relevant information about the execution of the components, but not to get overwhelmed with them. Also, from the software engineering perspective, it is important not to pollute the client code with too many logging specific calls, as it hurts the readability and maintainability. Basic Java logging solutions usually allow developers to configure the level of logs per classes, but they provide support neither for remote logging, nor for performance measurements. Therefore they were not usable for our purpose. A custom logging library was written instead. This library configures itself from the Enticing configuration DSL and has three types of supported destinations for log messages – stdout, file and a remote server. Each of these destinations can have different filters regarding log importance. Also, a support for performance measurements was added.

The main abstraction of this library is a *LoggerFactory*, which creates individual *Loggers*. A *Logger* can either be created with a specified name or its name can be inferred automatically from the surrounding class. Using a *Logger* instance, a function can write logs of various importance using the typical API and on top of that, it can wrap any piece of code and measure its execution. Internally, the logs are filtered and dispatched to all configured destinations.

8.3 Management and monitoring infrastructure

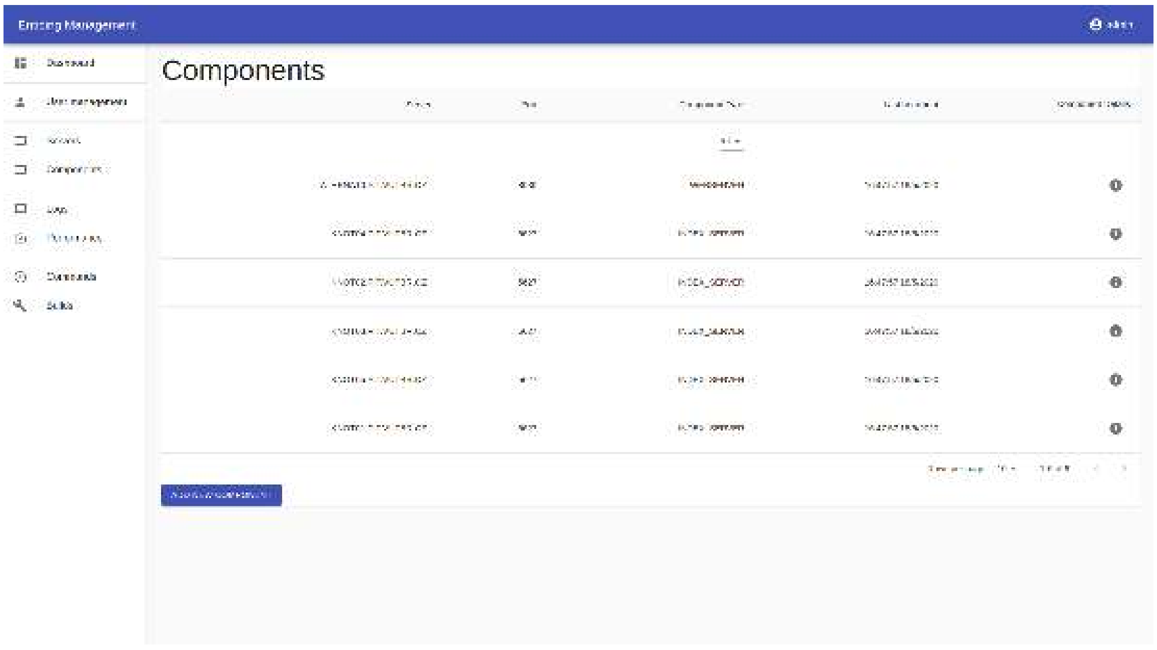
As the number of used servers increased, a need for a management component became apparent. It was very easy to loose track about which components were running on which servers and what was their current state. Therefore a management server was designed and implemented as another component written in Kotlin. This server receives important logs from all other components, so that the system administrators can easily monitor and manage the system from one place.

Each component of the system is given a URL to the management system. Once it is started, it registers itself and then starts sending heartbeats periodically. These heartbeats also contain information about the server the component is running on. During registration, static information such as the number of cores and the RAM size is sent. Then, current CPU and RAM usage is sent in each heartbeat message. As an example of the GUI, the table with running components can be seen in the figure 8.1.

Apart from the monitoring purposes, this server can also start and kill components. In the future, it can be extended to perform other maintenance tasks, some of them long-running, such as the distribution of *mg4j* files and their indexing. These long running tasks are modelled as commands and executed via special subsystem.

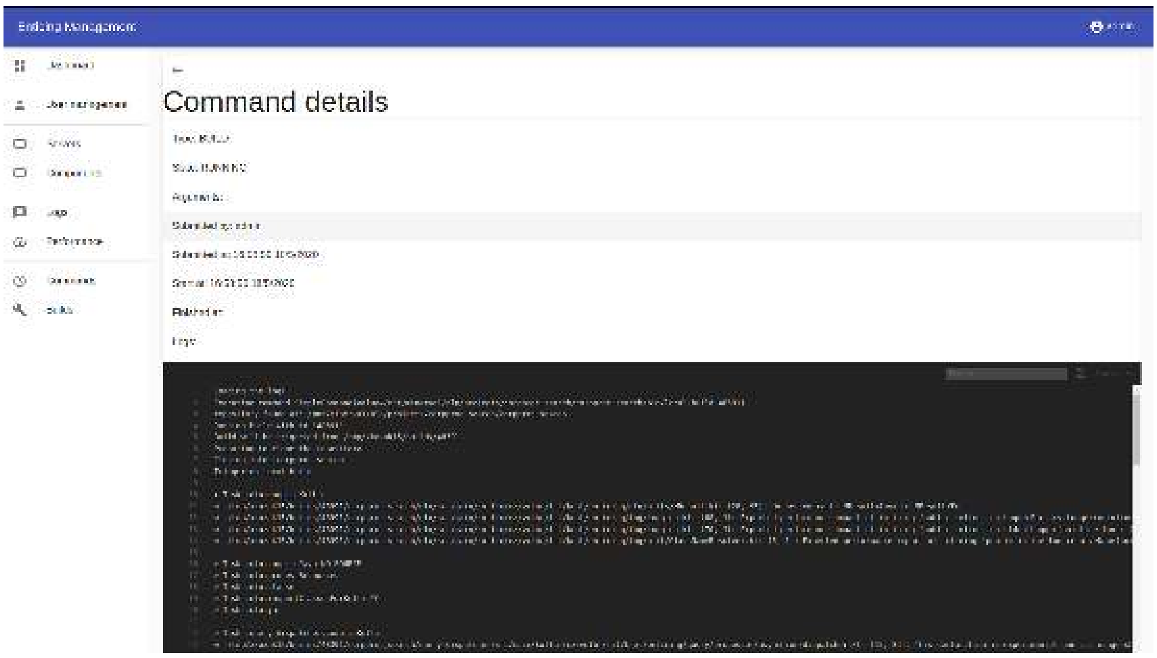
To allow for easier deployment of new versions of the platform, the management also supports its own *Continuous Deployment* pipeline, whose goal is to make the deployment as smooth as possible. It is possible to submit a *build* command, whose result are new jar files that can be used to start components. Screenshot of the running build can be seen in the figure 8.2.

This management server consists of three modules. The first one is *management-core*, which contains support for executing commands that start, monitor and kill other components of the platform. It can also be started as a command-line tool working in a very similar way to the previous solution based on Python scripts. But thanks to the fact that it is a Kotlin module, it can use the Enticing configuration script instead of the redundant



Components

Figure 8.1: List of running components in the management server



Command details

Type: BUILD

Status: RUNNING

Arguments:

Submitted by: admin

Submitted at: [illegible]

Started: [illegible]

Finished at:

Logs:

```
[illegible]
```

Figure 8.2: Running build command

.ini file. On top of that, it can be tested thoroughly and these tests can be included in the CI pipeline.

The second module, *management-server*, provides a REST interface over the *core* module supporting all the operations described above. It also persists currently known servers and components to the database, along with their heartbeats and logs. It was developed as a Spring Boot application. It also contains a command subsystem for execution and monitoring of commands submitted from the GUI.

The last module, *management-frontend*, contains a JavaScript single page application which serves as the GUI of the component. It was written in Typescript using React & Redux.

Chapter 9

Conclusion

The topic of this thesis was semantic searching over big textual data. The Knowledge Technology Research Group (KNOT)[1] at the Faculty of Information Technology Brno University of Technology (FIT BUT) has a Natural language processing (NLP) pipeline which can analyse documents written in natural languages and add additional meta information to them. Such information can be *syntactic*, such as lemma of the word or their position within sentences and paragraphs, or *semantic*, such as entities like people and places. The output of this pipeline is a big volume of textual data. It is already a great piece of work on its own, but without the ability to query these semantically enhanced documents, their usage is limited. The goal of this thesis was to design and develop a search engine that would query the documents efficiently while allowing to use all the meta information in the queries.

Firstly, the topic of indexing and searching inside search engines was introduced. Then MG4J[2], a search engine used internally in the resulting infrastructure, was discussed. Afterwards, followed the topic of semantic enhancement of natural languages. The corpora processing pipeline[3] used within KNOT to create semantically enhanced documents was described. Several state of the art search engines with support for semantic search have been analyzed along with their strengths and shortcomings.

After providing an overview of related theory, the design of the new search engine called Enticing was presented. It is a distributed system consisting of multiple components. The most important ones are the *IndexServer* for maintaining and querying slices of the data and the *Webserver* for dispatching requests to *IndexServers* and presenting results to the user. Each of these components consists of several modules with well-defined interfaces.

These components were designed to be deployed as separate processes on different servers. Therefore, the system handles most of possible exceptions in its components without shutting down totally and at least partial results are presented to the user whenever possible. On the other hand, it was necessary to design, implement and test the system very carefully, as the distributed nature created a lot of challenges. Some of the most interesting ones were discussed.

In order to query the semantic metadata, new query language called EQL (Enticing Query Language) was designed. This language is powerful enough to query all the entities inside semantically enhanced documents but also simple to understand, so that users from

[1]https://www.fit.vut.cz/research/group/knot/

[2]http://mg4j.di.unimi.it/

[3]http://knot.fit.vutbr.cz/corpproc/corpproc_en.html

other domains can use it as well. A compiler for EQL was implemented and integrated into the platform, which also included creating specialized searching algorithms based on EQL.

The platform was tested thoroughly, both for correctness and for performance, and then successfully deployed. The resulting system is now publicly available via the Internet[4]. It was presented at Excel@FIT 2020[5], where it received an Expert Panel Award. At the time of writing this text, it is running stable for days without any significant issues. The core was further extended by adding a monitoring infrastructure, maintenance system, configuration DSL, support for eager result loading and a smart search bar with syntax and semantic validation. It can be further extended by adding support for inheritance between entities, new types of *IndexServers*, for example backed by a neural network based question answering model, or by adding a native mobile client. Another interesting extension might be autocomplete support for the smart search bar.

[4]At the time of writing this thesis, it was deployed at `http://athena10.fit.vutbr.cz:8080/`
[5]`http://excel.fit.vutbr.cz/`

Bibliography

[1] Abramov, D. et al. *Redux – A Predictable State Container for JS Apps* [online]. 2020 [cit. 2020-03-05]. Available at: https://redux.js.org/.

[2] Aho, A. V., Sethi, R. and Ullman, J. D. *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[3] Boldi, P. and Vigna, S. *Mg4j (big) The Manual*. [cit. 2020-03-20]. Available at http://mg4j.di.unimi.it/man-big/manual.pdf.

[4] Boldi, P. and Vigna, S. *MG4J: high-performance text indexing for Java* [online]. 2005 [cit. 2020-03-15]. Available at: http://www.mg4j.di.unimi.it/.

[5] Boldi, P. and Vigna, S. Efficient optimally lazy algorithms for minimal-interval semantics. In: *Theoretical Computer Science*. Dipartimento di Informatica, Università degli Studi di Milano, Italy: [b.n.], 2016, p. 8–25. ISSN 0304-3975.

[6] Breslav, A. et al. *Kotlin programming language* [online]. 2020 [cit. 2020-03-27]. Available at: https://kotlinlang.org/.

[7] Doležal, J. *Komponent pro sémantické obohacení*. Brno, CZ, 2018. Master Thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/7848/.

[8] Grešová, K. *Searching Semantically Annotated Texts*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[9] Haverbeke, M. et al. *CodeMirror* [online]. 2020 [cit. 2020-03-09]. Available at: https://codemirror.net/.

[10] Hejlsberg, A. et al. *TypeScript – JavaScript that scales* [online]. 2020 [cit. 2020-04-03]. Available at: https://www.typescriptlang.org/.

[11] Hoeller, J., Deleuze, S., Long, J. et al. *Spring Framework* [online]. 2020 [cit. 2020-03-21]. Available at: https://spring.io/.

[12] Kilgarriff, A., Rychlý, P., Jakubíček, M. et al. *SketchEgine* [online]. [cit. 2020-03-23]. Available at: www.sketchengine.eu.

[13] Kozak, D. Enticing – Semantic Search Engine. *Excel@FIT 2020*. 2020, [cit. 2020-06-01]. Available at: http://excel.fit.vutbr.cz/submissions/2020/019/19.pdf.

[14] Panov, S. *Indexing and Searching Semantically Annotated Texts*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[15] Parr, T. et al. *ANTLR* [online]. 2020 [cit. 2020-03-12]. Available at: https://www.antlr.org/.

[16] Tablan, V., Bontcheva, K., Roberts, I. and Cunningham, H. Mímir: an Open-Source Semantic Search Framework for Interactive Information Seeking and Discovery. *Journal of Web Semantics*. 2014. Available at: http://dx.doi.org/10.1016/j.websem.2014.10.002.

[17] Vaughn, B., Abramov, D., Gannaway, D. et al. *React – A JavaScript library for building user interfaces* [online]. 2020 [cit. 2020-03-05]. Available at: https://reactjs.org/.

[18] Vigna, S. Quasi-succinct indices. In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. Rome, Italy: ACM, 2013, p. 83–92. ISBN 978-1-4503-1869-3.

Appendix A

Contents of the included storage media

The storage media contains a clone of the project git repository at the time of submitting the thesis[1], text of the thesis and a poster. The structure is as follows.

```
root
├─ repository ........................ Enticing git repository
│  ├─ bin ....... Scripts for starting individual components
│  ├─ console-client ................. ConsoleClient module
│  ├─ data ......................... Input data for testing
│  ├─ deploy .............. Deployment configuration files
│  ├─ documentation ............. High-level documentation
│  ├─ dto ................................ Core dto module
│  ├─ eql-compiler .................. Eql-compiler module
│  ├─ index-builder ................. IndexBuilder module
│  ├─ index-lib ......................... IndexLib module
│  ├─ index-server ................... IndexServer module
│  ├─ lib ...... Compiled components as jar archives and dependencies
│  ├─ management-core ........... Management core module
│  ├─ management-frontend ... Management frontend module
│  ├─ management-service ..... Management service module
│  ├─ query-dispatcher ........ QueryDispatcher module
│  ├─ scripts ............... Original management scripts
│  ├─ testings ..... Integration and performance tests
│  ├─ webserver-frontend .... Webserver frontend module
│  ├─ webserver ....................... Webserver module
│  └─ README.md ...................... Main readme file
├─ thesis
│  ├─ thesis-pc.pdf ........ PDF version for reading on a PC
│  ├─ thesis-print.pdf .............. PDF version for printing
│  └─ source ....... Directory with LaTeX source files of this thesis
└─ poster
   ├─ poster.pdf ................ PDF version of the poster
   └─ source ....... Directory with LaTeX source files of the poster
```

[1]Up to date version can be found on GitHub at https://github.com/d-kozak/enticing

Appendix B

Manual

B.1 Dependencies

To build and run the components, following dependencies are needed.

- Java – version 1.8 or higher
- Gradle – version 5.3.1 or higher

B.2 Build

To build the project, you can either use a script file in the *bin* folder or use gradle command directly. Both of the options below will work.

```
./bin/build
gradle buildAll
```

B.3 Testing

Unit tests

To run the unit tests, gradle can be used.

```
gradle clean test --info
```

Integration and performance tests

These tests are very expensive to run and cannot be performed from the CI pipeline, that's why they are disabled by default. To run them, open the testing module in the IDE of your choice and select wanted tests manually.

Appendix C

EQL Grammar

This chapter contains the grammar of EQL written in the Antlr4 format[1].

```
grammar Eql;

root: queryElem (CONSTRAINT_SEPARATOR constraint)? EOF;

queryElem:
    IDENTIFIER COLON EQ queryElem #assign
    | NOT queryElem #notQuery
    |(RAW |IDENTIFIER | ANY_TEXT | interval) #simpleQuery
    | IDENTIFIER COLON queryElem #index
    | IDENTIFIER DOT IDENTIFIER COLON queryElem #attribute
    | queryElem EXPONENT queryElem #align
    | PAREN_OPEN queryElem PAREN_CLOSE proximity? #parenQuery
    | queryElem booleanOperator queryElem proximity? #booleanQuery
    | queryElem LT queryElem proximity? #order
    | QUOTATION queryElem+ QUOTATION #sequence
    | queryElem queryElem proximity? #tuple
    ;

proximity : SIMILARITY IDENTIFIER ;

interval: BRACKET_OPEN (ANY_TEXT|IDENTIFIER) DOUBLE_DOT
(ANY_TEXT|IDENTIFIER) BRACKET_CLOSE;

constraint: booleanExpression;

booleanExpression:
    comparison #simpleComparison
    | NOT booleanExpression #notExpression
    | PAREN_OPEN booleanExpression PAREN_CLOSE #parenExpression
    | booleanExpression booleanOperator booleanExpression #binaryExpression
    ;
```

[1]See https://www.antlr.org/ for details

```
comparison: reference comparisonOperator referenceOrValue;

referenceOrValue: reference | nestedReference;

reference: IDENTIFIER (DOT nestedReference)?;

nestedReference: IDENTIFIER | RAW;

booleanOperator: AND | OR ;

comparisonOperator: EQ | NE | GT | GE | LT | LE ;

RAW: [’]~[’]+[’];

CONSTRAINT_SEPARATOR: ’&&’;

COLON:’:’;
DOUBLE_DOT:’..’;
DOT: ’.’;
EQ: ’=’;
NE: ’!=’;
GT: ’>’;
GE: ’>=’;
LT: ’<’;
LE: ’<=’;
EXPONENT: ’^’;
SIMILARITY:’~’;
SENT: ’_SENT_’;
PAR: ’_PAR_’;
NOT: ’!’;
AND: ’&’;
OR: ’|’;
PAREN_OPEN : ’(’;
PAREN_CLOSE : ’)’;
BRACKET_OPEN: ’[’;
BRACKET_CLOSE: ’]’;
MINUS:’-’;
QUOTATION: ’"’;

IDENTIFIER: [_]?[a-zA-Z0-9][a-zA-Z0-9_]*;
ANY_TEXT: ~[ !"’\u005B\u005D\t\r&|=<>:.()*^-]+[*]? ;

/** ignore whitespace */
WS : [ \t\r] -> skip;
```