Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Solution for Management and Visualization of Complex Search Queries

Daniel Chabr

Supervisor: RNDr. Jiřina Scholtzová, Ph.D.

11th May 2015

Acknowledgements

I wish to express my sincere thanks to RNDr. Jiřina Scholtzová, Ph.D., for providing me with her professional advice and support throughout the revisions of the thesis. I take this opportunity to express my gratitude to Ing. et Ing. Martin Švík, Ph.D., for helping me choose and specify the topic of the thesis. I also place on record my sincere thank you to Ing. Milena Rothbauerová for her kind help with proofreading. I also thank my parents for their unceasing encouragement and support. Finally, I am very grateful to my partner who has supported me through this venture.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 11th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Chabr, Daniel. *Solution for Management and Visualization of Complex Search Queries*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce se zabývá možnostmi usnadnění procesu vytváření komplexních vyhledávacích dotazů pro podnikové vyhledávací platformy. Analytická část práce objevila společné rysy současných vyhledávacích jazyků. Tyto společné rysy byly poté použity k návrhu řešení pro vizualizaci dotazů v přehlédné stromové struktuře a pro správu vyhledávacích dotazů. Poté byl implementován a otestován prototyp navrhnutého řešení. Implementovaný prototyp úspěšně demonstroval možné řešení zadaného problému.

Klíčová slova vizualizace dotazů, správa dotazů, vyhledávací dotaz, dotazovací jazyk, podnikové vyhledávání

Abstract

This thesis concerns the possibilities of facilitating the process of creating complex search queries for enterprise search platforms. The analysis part of the thesis discovered common patterns in contemporary query languages. These patterns were then used to design a solution for visualizing the queries in a lucid tree structure and for the management of the queries. Then, I implemented a prototype of the designed solution and tested it. The implemented prototype resulted in a successful proof of concept.

Keywords query visualization, query management, search query, query language, enterprise search

Contents

List of Figures

List of Tables

Introduction

Enterprises have an abundance of data from various sources. To make use of the data, they utilize enterprise search platforms (ESPs) that enable the users to perform searches across all the sources and provide the results in the same user interface. The ultimate goal of an ESP is to respond to the user's search query with the most relevant results. There are many tools that help understand the obtained results of a query and there has been great focus on improving underlying algorithms and the infrastructure of ESPs, but very little has been done to examine the possibilities of facilitating the user to build more accurate queries. All the major ESPs on the market offer extensive query languages for performing accurate searches. But the syntax of these languages is too difficult for common users.

Aim of the thesis

The aim of this thesis is to design a solution that would let even non-expert users to perform advanced queries for greater result relevancy. This is to be achieved via the visualization and the management of queries. But first, the analysis of query languages (QLs) needs to be done in order to prove that QLs have a common structure that can be visualized. These patterns will be then used to design a solution for visualizing the queries in a lucid tree structure and for the management of the queries. Based on the designed solution, I will describe the process of implementing and testing the prototype for IBM Watson Content Analytics (IWCA) ESP. At last, I will describe necessary steps for adapting the prototype for other ESPs.

Chapter 1

State-of-the-art

1.1 Enterprise search platforms

Anyone with access to the internet has to use some form of a web search technology to access the desired data. If I had to browse through websites one by one to find what I need, it would take ages. Web search makes any website accessible within seconds provided I know keywords that are specific enough. In these terms the enterprise search is very similar to web search but there are also key differences.

The main difference is that the enterprise search makes accessible primarily data coming from intranet, that is, data stored on a network not directly connected to the internet [3].

Next is that the enterprise search (ES) supports the so-called hybrid search which lets users perform searches not only across various documents but also across database records because any differences are abstracted away by the platform [3].

Another difference is the number of sources of data. The enterprise data is often stored at multiple repositories with various underlying architecture. The user's search goes through all the sources without the user even noticing because the ESP can connect to each source's interface [4]. This is achieved by the use of connectors and it is called a federated search.

These are the principal differences from the web search but there are other like the high security standards or the incremental search, that enables a partial update of the index without the need to rebuild the whole index from the start when the data changes. Actually there is one more difference that is of interest for the purpose of this thesis. The Google web search engine uses PageRank, an algorithm for objectively rating web pages largely based on the number of referring web pages. This algorithm has proved to be very accurate in estimating the web page's relevancy and it is a big reason of Google's success [5]. Other web search engines are likely to use very similar methods. But this method cannot be used in the ES because company data

is not browsed by users randomly creating links between documents based on their personal opinion [3]. The search engine's ability to presume what the user is looking for is thus limited. This poses bigger demands on the user's queries.

One way of aiding the user to make more precise queries is a faceted search. *The faceted search interface allows the user to progressively narrow down the choices by choosing from a list of suggested query refinements* [6]. This is a useful tool when searching through structured data. Title, author, date or even Global Positioning System (GPS) coordinates are good search criteria if I can identify them within each document. But what if the search engine cannot identify these fields in each document? What if the data is unstructured and only the end user can make assumptions about the contents of the desired document?

The main goal of ESPs is to take user's simple query and return results sorted by relevancy [7]. This would be the perfect case, but often a simple query is not enough and in that case the user is expected to specify the search query more accurately to narrow down the results. This is the very purpose for which the ESPs provide QL support.

1.2 Enterprise search query languages

Creating a search query in an ESP can be as easy as entering a keyword or as complex as using several specific types of constraints on tens or even hundreds of keywords. The latter option would be available to advanced analysts only. Let's take a look at an example of a complex search query for Apache Solr platform.

```
(title:oak^1.5 OR title:birch*) (body:tree OR body:log) AND
"mine wood"~3 mod_date:[20020101 TO 20030101]
```

I might guess the meaning of some parts of the query but other remain almost cryptic without further knowledge. This example follows the syntax of a QL of an open source Apache Solr platform. The query would likely return results with the title field containing the word `oak` or a word beginning with `birch` while the results with `oak` in the title would take priority in the order of results. The body of the results would have to contain at least one occurrence of the word `tree` or `log`. Anywhere in each result would have to be words `mine` and `wood` with the maximum of three other words in between them. All the results would also have to be modified for the last time in the year of 2002. All of that of course is dependent on whether such results even exist in the searched sources and the behaviour of the query can also be changed by configuration of the platform. This query only scratches the surface of what some QLs allow. Every ESP has its own characteristic query language, but even though the syntax may vary greatly, the meaning is usually very similar.

Chapter 2

Analysis

In this chapter I will explore requirements of the implemented solution, do a deeper research of QLs and finally I will point out any existing solutions.

2.1 Requirements

Requirements of the solution, that is the subject of this thesis, have been continually discussed with the IBM experts specializing in the enterprise search technology so that the results of this work have practical benefits.

Functional requirements

- graphical user interface (GUI) for building queries
- visualize query in an understandable way
- save any query or self-contained part of the query under alias for later reuse
- edit or delete saved queries
- export built queries in a valid enterprise search query syntax
- import queries written in an enterprise search query syntax
- save and provide history of queries
- support a view of arbitrarily long queries

Non-functional requirements

- web interface
- ESP independence

2.2 Query languages

To understand the enterprise search QLs and the possibilities of working with them, first, I had to research their functionalities and common structures. Unfortunately there are no general guidelines for QLs and their syntax. The best source of knowledge in this topic is the documentation of the QL for each specific ESP.

2.2.1 Enterprise search market leaders

To narrow down the field of research I have made a selection of market leaders that would make up a representative sample for my further research. I have found out that Gartner, Inc., the technology research and advisory company, makes an annual report on the enterprise search industry in the form of a graph (see figure 2.1) [1]. Based on this report, the availability of public QL documentation and the professional advice given to me, I have chosen several ESPs whose documentation would serve as a source for my QL research.

Selected ESPs:

- open source – here, the choice was easy, as the two platforms are clear leaders in the open source market
  - **Apache Solr** [8], [9] is a traditional choice and it has a big developer community
  - **Elasticsearch** [10] is a very popular enterprise search platform for its ease of use
- commercial
  - **Coveo Enterprise Search** [11] was chosen based on the figure 2.1
  - **Google Search Appliance (GSA)** [12] was chosen based on the figure 2.1
  - **HP Autonomy** [13] was chosen based on the figure 2.1
  - **IWCA** [14] was chosen for its strong analytic capabilities and for being a traditional enterprise choice. Furthermore, the implemented prototype will be deployed and tested on IWCA thanks to IBM, that provided me with their advice and with the access to their software.
  - **Microsoft SharePoint** [15] was chosen because it is deployed in many businesses and even though it serves primarily as a cloud solution, it supports a rich query language

Figure 2.1: Gartner 2014 Magic Quadrant Report [1]. The quadrant is divided into four parts, the top right quadrant identifies leaders, that execute well against their current vision and are well positioned for the future. The bottom right quadrant identifies visionaries who understand where the market is headed but lack in execution. The top left quadrant identifies challengers that execute well today but are not prepared for future market changes. Finally the bottom left quadrant sees niche players that are successful in a small segment but do not out-innovate or outperform others. [2]

2.2.2 Features

Following the review of documentation of my sample of ESPs, I was able to extract the common syntax features. Here are the features with their explanation and an example. Query examples are modified to demonstrate the described features and will not function in some ESPs in the stated form. Most of the syntax is taken from the Apache Solr platform.

wildcard search matches documents containing a word matching the wildcard expression (the wildcard character matches arbitrary string), e.g. `app*` would match "apple" as well as "appendix"

fuzzy search matches documents with the same or similar term to the searched term, e.g. `man~` would match "man", "men" but also "manly"

exact phrase search matches only documents containing exactly the same phrase, e.g. `"red apple"` would not match "apple red" or "red apples"

proximity search matches documents with stated terms within a specified distance from each other, e.g. `(apple pear) NEAR 3` would match documents where terms "apple" and "pear" are at most 3 words apart

field search matches documents containing a field that matches the specified expression, e.g. `author:John` would match documents where the author field contains "John"

range search matches documents with the field value within a specified range, e.g. `modified_data:( -week TO now)` would match documents modified in the last week

boolean search lets the user connect multiple search terms with operators, `AND` (this is the usual default implicit value which means that when I input a query like `apple pear` the engine understands it as `apple AND pear`) and `OR` operators are usually available, e.g. `apple OR pear` matches documents containing at least one of the two terms

term boosting is used with multiple search terms and lets the user boost relevancy of some terms, e.g. `title:apple^2 body:apple^1` would favour results where "apple" is in the title field over those where it is only in the body field

exclude term search matches documents that do not contain the specified term, e.g. `apple NOT pear` would match documents with "apple" **and** without "pear"

grouping enables the user to group expressions to form subqueries for more complex querying, e.g. `apple OR (pear AND cherry)` would match anything containing either "apple" **or** "pear" as well as "cherry"

regexp search matches documents where a string of characters matches a regular expression[1], `RegExp([a-dx]zz)` would match documents containing any of the following strings: "azz", "bzz", "czz", "dzz", "xzz"

2.2.3 Research of feature support

My research of the documentation yielded the table 2.1 of a feature support by each ESP vendor. I can see that six out of eleven features are supported by all of the researched ESPs. I will use this knowledge later when designing the solution.

Features	*Solr*	*ElasticSearch*	*Coveo*	*GSA*	*Autonomy*	*IWCA*	*SharePoint*
Wildcard s.	✓	✓	✓	✓	✓	✓	✓
Fuzzy s.	✓	✓	✓	×	✓	✓	×
Exact phrase s.	✓	✓	✓	✓	✓	✓	✓
Proximity s.	✓	✓	✓	×	✓	✓	✓
Field s.	✓	✓	✓	✓	✓	✓	✓
Range s.	✓	✓	×	✓	✓	✓	✓
Boolean s.	✓	✓	✓	✓	✓	✓	✓
Term boosting	✓	✓	×	×	✓	✓	✓
Exclude term s.	✓	✓	✓	✓	✓	✓	✓
Grouping	✓	✓	✓	✓	✓	✓	✓
RegEx s.	×	✓	✓	×	×	×	×

Table 2.1: Comparison of the supported features by vendors. All information in the table is based on publicly available documentation and may not be fully accurate. Only the out of the box functionality is considered. Symbol ✓means that the ESP supports that feature, symbol × means it does not.

[1]Regular expressions let the user define complex rules for matching character strings. Regular expressions may vary based on implementation and feature support.

2.3 Existing solutions

After investigating any query building tools, I have discovered that there are quite a few of them focused on building SQL queries, such as the following: Microsoft SQL Server Query Builder [16], DbVis Software Query Builder [17], SQLeo Visual Query Builder [18], dbForge Query Builder for SQL Server [19], Easy Query Builder [20] and Active Query Builder [21]. These tools are very advanced for their purpose, however, their narrow specialization on SQL queries makes them unusable for enterprise search queries.

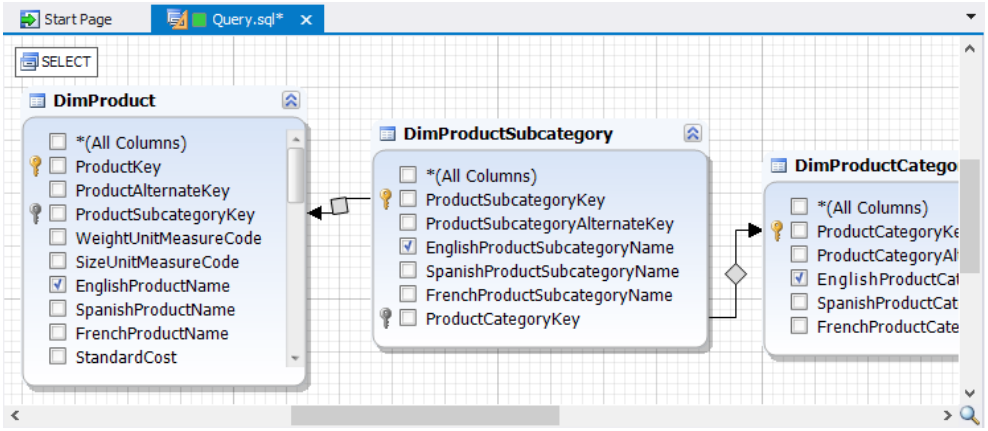

Figure 2.2: dbForge Query Builder's GUI is tailored to the needs of the SQL. The diagram shows a visual way of creating a query with three joined tables and three selected columns.

2.3.1 jQuery Query Builder

In April 2014, a developer named Damien Sorel started developing a jQuery Query Builder [22], a tool for building complex queries. I find it the best solution of its kind so far. The GUI of the tool is on figure 2.3. The code is easily extensible, so a custom export into arbitrary QL can be developed. However, for the purpose of this thesis, it meets only part of the requirements. It is missing a server part which would provide persistency of the queries. It lacks any way of importing queries, thus it cannot be used for modification of previously built queries, but only to build new ones from the very beginning. Furthermore, while it provides good query building options, I can imagine a cleaner GUI that would also help a user to understand the queries better, for example through visualization.

I started the implementation part of the module designed in chapter 3 at the end of September 2014. When I was doing research of existing solutions, the jQuery Query Builder was not yet publicly available. Even though I have chosen a different approach, it is interesting to see a similar solution arise at the same time. I believe that I might have taken some inspiration from jQuery Query Builder had I known about it at the time of designing the solution.

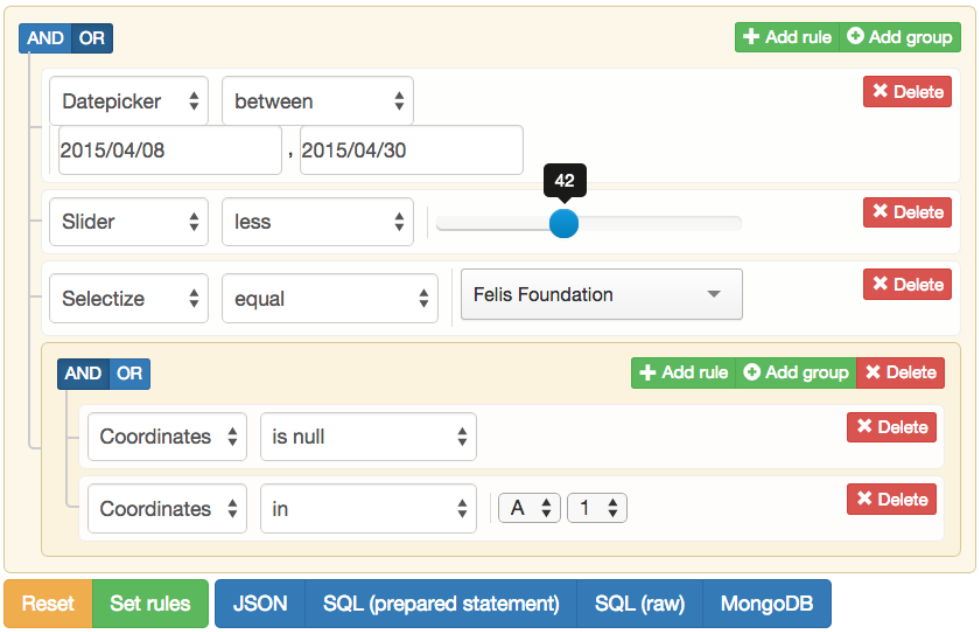

Figure 2.3: GUI of the jQuery Query Builder

Chapter 3

Design

In this chapter I will design a query building module that will meet all the previously stated requirements.

3.1 Architecture

As stated in the non-functional requirements, the module should provide a web interface for interaction. This is a sensible requirement which will make the module independent of ESP and of any operating system. Both previously analyzed open source ESPs, as well as for example IWCA, provide a web interface and ways of adding plug-ins to it. The web interface of the module will be integrated as such a plug-in.

The greatest advantage of web applications is that they are accessible from any device with a web browser independent of the operating system running on the device. All modern browsers provide at least three ways of storing the data. These are the session storage, the local storage and cookies. Cookies are intended for only small amounts of data up to 4 kilobytes and are transmitted with every request to the server. That could easily become insufficient for the purpose of storing tens of complex queries. The session storage in most browsers provides space for 5 megabytes of data, which would be satisfactory, but as its name suggests, the session storage lasts only as long as the session is valid. This means that any stored data gets deleted when a user navigates to a different domain, closes the tab or the browser itself. The last one mentioned, the local storage, can hold the same amount of data as the session storage and it does not have any expiration date. But even local storage is bound to the concrete browser and does not allow the user to continue his work for example when he comes home. This means that the requirement of saving queries for later reuse has to be solved by persisting the queries on the server part. On

the other hand the history of queries is more intended to avoid losing the progress by accidentally closing the browser or to step back from unsuccessful attempt at modifying a query. For this purpose the local storage is perfect as it also restricts the history to the current user.

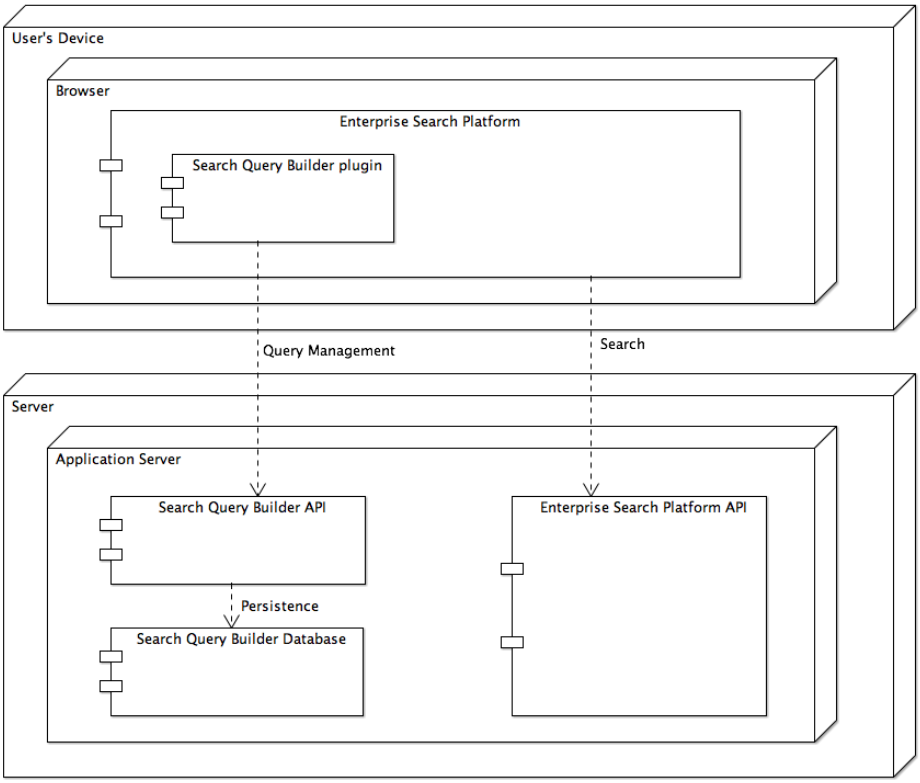

Figure 3.1: Deployment diagram

Enterprise software usually runs on application servers that take care of the efficient use of resources and provide the security and configuration management. Out of the seven previously analyzed ESPs, four are implemented in Java, two are implemented in .NET and the GSA is sold and deployed on its own hardware, so the implementation language is unknown.

There is no need to install and maintain another environment for the module. Instead, I can deploy it on an application server right next to the ESP. This will also avoid any problems with the browser's built-in same origin policy which restricts data to be loaded from a different domain or port. This and all the previous conclusions result in the deployment diagram 3.1.

3.1.1 Backend

For the remainder of this thesis, I will refer to the code run in a browser as the frontend and to code run on a server as the backend. The backend part of the module will provide the management of queries. The query entities will be accessed over the Hypertext Transfer Protocol (HTTP) and persisted to a database. The class diagram 3.2 shows a basic backend that should meet all basic persistence requirements. One thing worth explaining is the difference between the queryString and queryParsed attributes of the Query class. The queryString will save an unchanged query string in the same form as it is sent to the ESP. In this state, the query would not be understandable by the module. For this purpose, it needs to be parsed to understand the meaning of each part of the query. This parsed query can be then modified by the user and maybe even enriched by additional data that would be lost in the plain query string otherwise. So, not to lose the user's progress while building a query, I will also persist the parsed form.

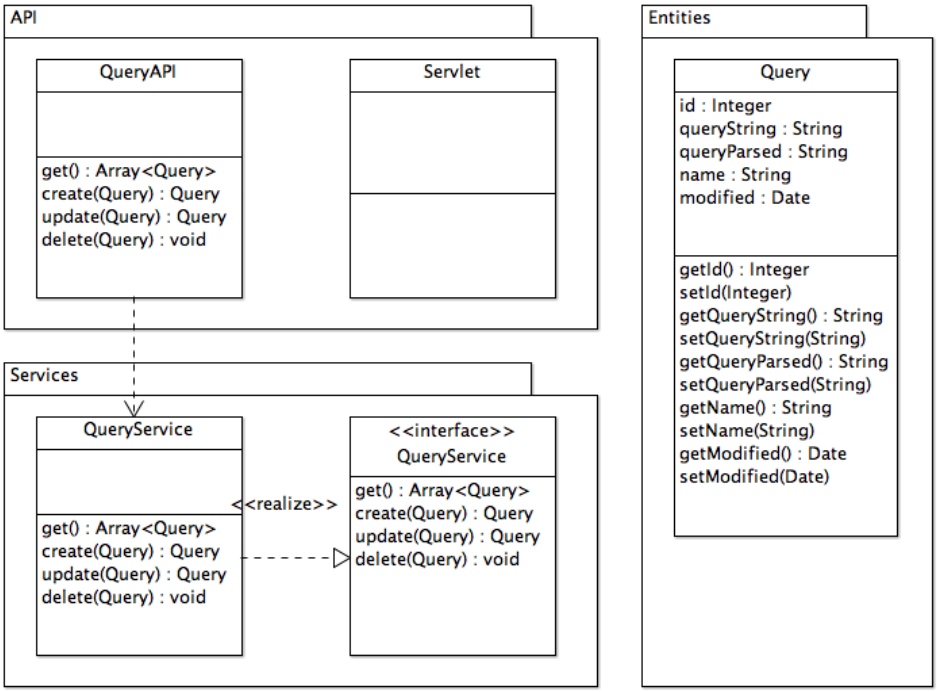

Figure 3.2: Backend Class diagram

The backend will perform the basic create, read, update and delete (CRUD) operations. I have decided to use the two layer architectural pattern that splits the application logic into a presentation and a data layer, as there was no need to use three layer pattern that adds a business layer for data manipulation.

The database will persist query objects with attributes identical to those of the Query entity in the diagram 3.2.

3.1.2 Frontend

As decided above, the frontend part will run in a browser, which means the code will have to be in the JavaScript programming language. The JavaScript code tends to get very chaotic if it is not taken care of and if it is not refactored periodically. To avoid this problem as much as possible, I have decided to use the Model-View-Controller (MVC) software architectural pattern, which divides responsibility into three separate parts. The model handles the data and notifies the view when the data changes. The view presents the data to a user and notifies the controller when a user's interaction takes place. Lastly, the controller decides what to do with the user's action and notifies the model to change the data, if necessary. This is a simple explanation and there are many variations to the MVC pattern.

In the diagram 3.3 the API package serves as a model, the PersistenceUI package as a view and the SQB package as a controller. Strictly speaking, JavaScript does not have a system of packages, but this diagram is for demonstrative and design purposes.

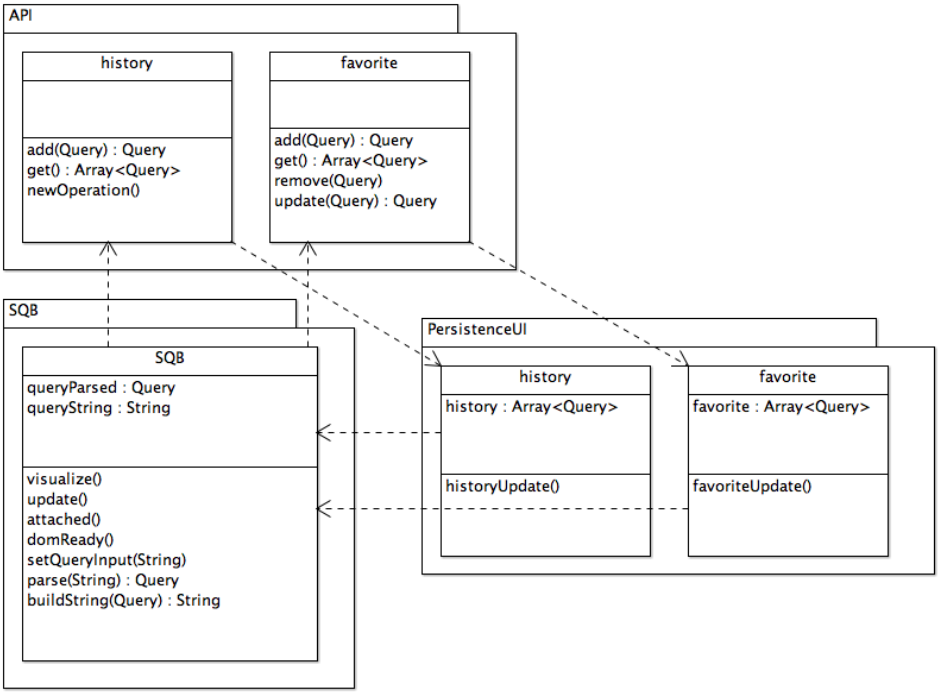

Figure 3.3: Frontend Class diagram

3.2 Parser

The module needs to understand the syntax of a chosen ESP to be able to visualize and modify the query. This is done by parsing the query and returning a structured object. The ESP has to do this too, but unfortunately it is done on the backend, so the plug-in cannot access the parsed object. This means that the module needs to have its own parser specific to the ESP's query syntax. Fortunately, there are open source parser generators available that, when given a grammar, return an optimized parser that can then be used to parse any query.

There are very many parser generators and the first criterion was the programming language that the output parser would be in. I wanted the plug-in to be fast and responsive to user's actions and modifications, so I decided to choose a JavaScript parser generator, which will eliminate the delay of a request to the server and response from the server, where the parser would have to be placed otherwise. The potential traffic delay outweighs any performance gains of running the parser in a compiled code on the backend mostly thanks to the JavaScript engine optimizations in the recent years.

As of this day, there are two mature parser generators for JavaScript, the Jison [23] and the PEG.js [24]. When I tried to write a simple query parsing grammar for each of them, it became clear that Jison would suit my needs much better thanks to its grammar's support for definition of an operator precedence and even for definition of an implicit operator.

Jison is mostly a JavaScript port of GNU Bison [25] which is a successful parser generator for C, C++ and Java programming languages. The grammar for Jison is very similar to Bison, which makes it very easy to find a good documentation and resources, because Jison is lacking in this area for now. Next, I had to decide on a form and structure of the parsed object. In case of JavaScript the form is very easy to decide on, as the JavaScript Object Notation (JSON) is natively supported by JavaScript, which makes easy to work with further on.

The most important language feature that will determine the form of the parsed object are the operators, which are present in all ESPs as discovered above. Each operator constitutes a connection between what is to the left and what is to the right of it. From now on, I will refer to such self-contained part of a query that can lie on either side of an operator as the node. I also know that all ESPs support grouping, which can cause that an operator with its nodes can become itself a node to another operator. All this is beginning to appear like a tree structure and I do intend to form the returned object into a tree structure. Additionally, each node that is not an operator, can have other properties that will store additional syntax information. Following is a query string in IWCA query language and the final parsed object in JSON.

The JSON separates the key from the value by a semicolon and nested objects are identified by curly braces.

```
title:((credit debit card) ANY 2) IN agent AND
(NOT account::((not interested) WITHIN 3 INORDER)
 IN client OR leave)
```

```
{
     "operator": "AND",
     "left"    : {
          "term"   : "credit debit card",
          "any"    : "2",
          "context": "agent",
          "field"  : "title"
     },
     "right"   : {
          "operator": "OR",
          "left"    : {
               "proximity": "3",
               "inorder"  : true,
               "term"     : "not interested",
               "context"  : "client",
               "facet"    : "account",
               "not"      : true
          },
          "right"   : {
               "term"     : "leave"
          }
     }
}
```

3.3 Visualization

To be able to build complex queries, the user needs to understand the meaning of it, so that he can foresee what documents might be returned and then optionally fine-tune the query later. The usual representation in a form of a plain text quickly becomes unsatisfactory when a few operators and a grouping takes place. As previously discussed, the structure of a parsed query forms a tree, more specifically a binary tree, as each node will always have either zero or two children. Tree data structures can be visualized in an expressive way, that is my goal with queries as well.

There are three main approaches at visualizing data in a browser. Firstly, the data can be visualized on the backend, where there is abundance of available tools, and a rendered image would be sent to the browser. Secondly, the visualization could take place in a Java applet or an Adobe Flash, both of which require a plug-in to be installed in a browser, and many mobile browsers do not support that altogether. The last option and the one I have chosen,

is to depend solely on what technology the browsers provide me with. Fortunately all up-to-date browsers support both JavaScript and Scalable Vector Graphics (SVG), which gave rise to several good visualization libraries that abstract away most of the differences between browsers and provide elegant application program interfaces (APIs) to visualize and manipulate data.

When deciding which library to use, I found a very nice example of a tree visualization [26] in the D3.js [27] library that suited my needs. Additionally, I was already familiar with using the D3.js library which made the decision easy.

3.4 User interface

Indispensable part of each ESP's GUI is a query input field. When a user writes a query in the input field and clicks the "Search" button, the query gets sent to a corresponding API on the ESP's backend. As the API in various ESPs may differ greatly, both in their payload structure and the uniform resource identifier (URI), I do not intend to replace the input field by the query builder's GUI, because there would have to be done too much configuration to make the query builder self-contained in executing the queries. Instead, the query builder plug-in can connect to the original input field and modify its contents when the user changes the query in the builder. It can also load any query modified in the input field and visualize it in the builder for further modification.

The dominant part of the query builder will certainly be the visualization. There are two main options for programmatically creating graphics in a web browser. There is the HTML5 canvas element and the browser SVG implementation. D3.js can visualize data in both of them, but SVG allows easier interaction with individual displayed objects, which I want to utilize for the query building purposes. The idea is that when a user clicks on any node, a context menu would be displayed to let the user add new nodes, modify the existing one or delete it. By this interaction the user would simultaneously modify the query string in the query input field. I have discovered a small D3.js extension that supports the context menu creation called d3-context-menu [28], which I will also make use of.

The last thing that requires a user interface is the query management. It will consist of two sections, one for the saved queries and one for the history. Both will provide a simple list of queries with the difference that the saved queries will be labeled with their assigned title, it will be possible to delete them and also there needs to be a way to name and save the queries.

3.5 Technology and implementation language

3.5.1 Backend

As the implementation language for the backend I have chosen Java, because as mentioned above, it is the most common implementation language for the ESPs and it is also platform independent, which means that even in case the module was to be integrated with Coveo or SharePoint, which both run on .NET framework, it would still be possible to install the Java query building module on the same server. For the module to be able to listen to requests from the web browser, it has to be either deployed on an application server that runs a web server or it would have to listen to the requests on its own. The second option could cause several problems with security, due to listening on the same port as the ESP or with the same origin policy. To make the solution as modular as possible, it will implement the Java Enterprise Edition (Java EE) standards, which will enable it to be deployed to any Java EE compliant application server.

3.5.2 Database

The next step was to decide how to persist the data on the server side. The usual way would be to install a database and configure Java database connector to connect to it. I did not like this option because it meant yet another unnecessary dependency, which would raise the maintenance costs. The perfect solution was to use an embedded database that gets packaged inside an application and does not need to be installed or maintained. The Apache Derby database management system offers just that, it is open source and well documented.

3.5.3 Frontend

On the frontend, I am limited by technologies that are supported by the web browsers. The basis forms the common trio of HyperText Markup Language (HTML) for defining elements, Cascading Style Sheets (CSS) for styling them and JavaScript for interacting with them and for the overall frontend application logic.

When sending requests from the frontend to the backend, I want to avoid reloading the whole page. For this purpose, I will use asynchronous JavaScript and JSON (AJAJ), a variant of the more commonly known asynchronous JavaScript and XML (AJAX). AJAJ allows the requests with JSON payload to be sent asynchronously without distracting the user and when a response comes back, the remaining necessary code gets executed, still notifying the user only when necessary.

To define understandable interface for communication between the front-end and the backend, I decided to use a set of architectural guidelines called Representational State Transfer (REST). Its main use is to design common interface for CRUD operations by defining a set of URI patterns and the corresponding HTTP methods. An API that follows these guidelines is usually referred to as RESTful API.

Operation	HTTP method	URI examples	Payload
Get all	GET	example.com/queries	no
Create	POST	example.com/queries	yes
Update	PUT	example.com/queries/{ID}	yes
Delete	DELETE	example.com/queries/{ID}	no

Table 3.1: A REST API for query persistence operations. The string {ID} would be replaced by an actual identification number of a query.

When the module receives a list of user's saved queries, it needs to present them to the user. For this purpose, there is a Document Object Model (DOM), a programming interface for modifying contents of a web page without reloading it. For creating AJAJ requests and to modify the browser's DOM, I will make use of a JavaScript utility library called jQuery [29], which defines many useful methods that will not only simplify my code but also make it less prone to errors.

Contrary to saving favorite queries, it is desirable that the history of queries is persisted in the browser. I have already described differences between the local storage, the session storage and the cookies in the section about architecture. I have decided to use basil.js [30], a minimal persistence layer that provides convenient methods for storing data. It provides me with a useful feature allowing to set my preferences for storing data and if the first option, for example the local storage, fails for some reason, basil.js still tries to persist the data to the session storage or as a cookie.

The reasons for using D3.js library and its d3-context-menu extension are described in the sections on visualization and user interface.

Chapter 4

Implementation

The aim of the implementation was to develop a functional prototype that would prove the benefit of the query building module. The prototype was developed and iterated over in cooperation with the IBM experts in the enterprise search field. The target platform for the implementation was the IWCA. In this chapter, I will discuss the undergone process of the implementation and I will pinpoint interesting or difficult to solve problems that I encountered. The implemented solution will be referred to as the search query builder or simply, the query builder.

The steps for setting up a development environment are described in the Programmer's manual (see Appendix A). The frontend is intended to be served from a web server, while the backend is intended to be deployed on an application server. This means that the enclosed implemented prototype cannot be run without installing the necessary tools.

4.1 Version management

To track the progress and any changes in the project, I used Git [31]. It allowed me to discard code experiments with ease, which I used at several occasions.

4.2 Project generator

Since there are several things to set up before one can begin developing for the frontend, I used Yeoman [32]. It offers a number of web project generators for various purposes. Each generator usually creates a sensible file structure, sets up a web server, installs necessary dependencies and creates a testing environment. There are currently over a thousand available generators for various purposes, but I could have also developed my own generator, if I did not like any of the existing ones.

4.3 Parser

The foremost thing that had to be developed, was the parser. If there was no parser, there would be no way to visualize or modify a query. Based on the design, I created a grammar for parsing IWCA's query language and then used Jison parser generator to create the parser. The most difficult thing was not to define the grammar, but to understand all possible syntax combinations. The documentation for IWCA's query language was good in terms of describing the syntax for various features, but it lacked examples of combining these features. For example, there is a way to define a field search with a semicolon like this `<field>:<term>` and there is also a way to define a facet search with two semicolons like this `<facet>::<term>`. So if I define a query like `title:bank::credit`, it is hard to estimate how the engine would understand it. In this case, only the last semicolons are interpreted as a syntax so it would search for a `credit` in a facet `title:bank`. Some of these conflicts were resolved by testing the queries and examining the results, while other conflicts were solved following the advice of experts with the knowledge of how the engine works.

4.4 Dependency management

The IWCA's frontend uses Dojo [33] framework to organize the code, manage plug-ins and a lot more. It uses efficient dependency management to define what each part of the code needs for its own execution. Because Dojo's dependency management uses the same API as a separate JavaScript tool called RequireJS [34], it can be used separately from Dojo as well. Since the query builder itself has several dependencies, I implemented the dependency management in the solution to avoid loading some dependencies multiple times which could very well happen with some popular libraries like jQuery.

4.5 User interface

The final user interface is in figure 4.1. It meets all the requirements set up in the design chapter. On startup, it can be configured to connect to other query input field already present in the ESP's GUI. In that case the query input field at the top of figure 4.1 would be hidden.

I have implemented four different ways for modifying a query. The first one is to modify the original query input. Any change to the query input field is propagated and the changed query is immediately visualized. Secondly, when a user selects any node of a visualized query by clicking on it, the query string of that part of the query is displayed in an input box labeled the "Active node". This string can be then modified as necessary and when a user presses the associated "Update" button, the active node is replaced by anything the

user input. This option can also be used to either delete the active node by deleting the contents of the input field or to add new nodes by filling in a query that contains new operators. The third option for modifying queries I added, is to click on any visualized node with a right mouse button. A contextual menu is displayed with options to add new operators, negate the node or to delete it. The last way of modification is to drag any node to a different position. When a user starts dragging a node, it is removed from the tree and when the node is dropped on an existing node, the user is asked what operator to use for the new connection.

The lists of saved queries and of the history are hidden by default and show only when the user moves a mouse over the title for that section. This is also visible in figure 4.1, as the saved queries are visible but the history is hidden. This assures the lists do not take up too much space when not needed.

4.5.1 Label placement

Because IWCA has its interface for adding many terms to the query with a simple `AND` operator, I was required that the visualization handles well queries with tens or even hundreds of queries. But the problem was that when I tried to visualize such query, the node labels often overreached each other and thus became unreadable. While trying to find a solution to this problem, I tried two approaches before settling on the third one.

Force-based label placement

The first approach was based on the possibility to apply forces to nodes in D3.js [35]. The theory was that if I apply repulsive forces on text labels and set a maximum distance from their nodes, the forces would get applied until the model would come to a balanced state of forces. Unfortunately, when I implemented it into the solution, it became clear that I would have to try a different approach. Firstly, it took about twenty seconds for thirty nodes to reach a balanced state, which was unacceptable, and secondly, even then the result was not satisfactory (see figure 4.2).

Constraint relaxing

The second approach I tried was to place the labels as usual and then check if they overlap. If they did, I would move them away from each other a little and check again until no labels overlap [36]. This approach is called the constraint relaxing. Even though this approach yielded slightly better results than the previous one, it required even more computational power and was more dependent on the number of nodes.

Tree size adaptation and zoom

The final solution to this problem was to compute the width of the longest label and set the distance between each level to be at least the same (see the top part of figure 4.3). This alone would not be enough, as large trees would not fit into the view or would be too small to effectively work with. Fortunately, D3.js supports zoom feature that allows not only to zoom in and zoom out the view, but also to drag it. A user can center any part of the tree that needs to be inspected in detail by dragging the view (see the bottom part of figure 4.3).

4.6 Backend integration

I developed the query builder separately and deployed only the major versions, because IWCA is a large application, which brings about high hardware requirements. To make the environment as similar as possible I used the IBM WebSphere Liberty Profile application server to run the query builder backend on my computer before deploying it to IWCA that runs on the same application server.

4.7 Frontend integration and configuration

The last step was to integrate the developed frontend plug-in into IWCA. Even though this step is not as straightforward as I presumed, the documentation [37] is very detailed and after several attempts everything started working seamlessly.

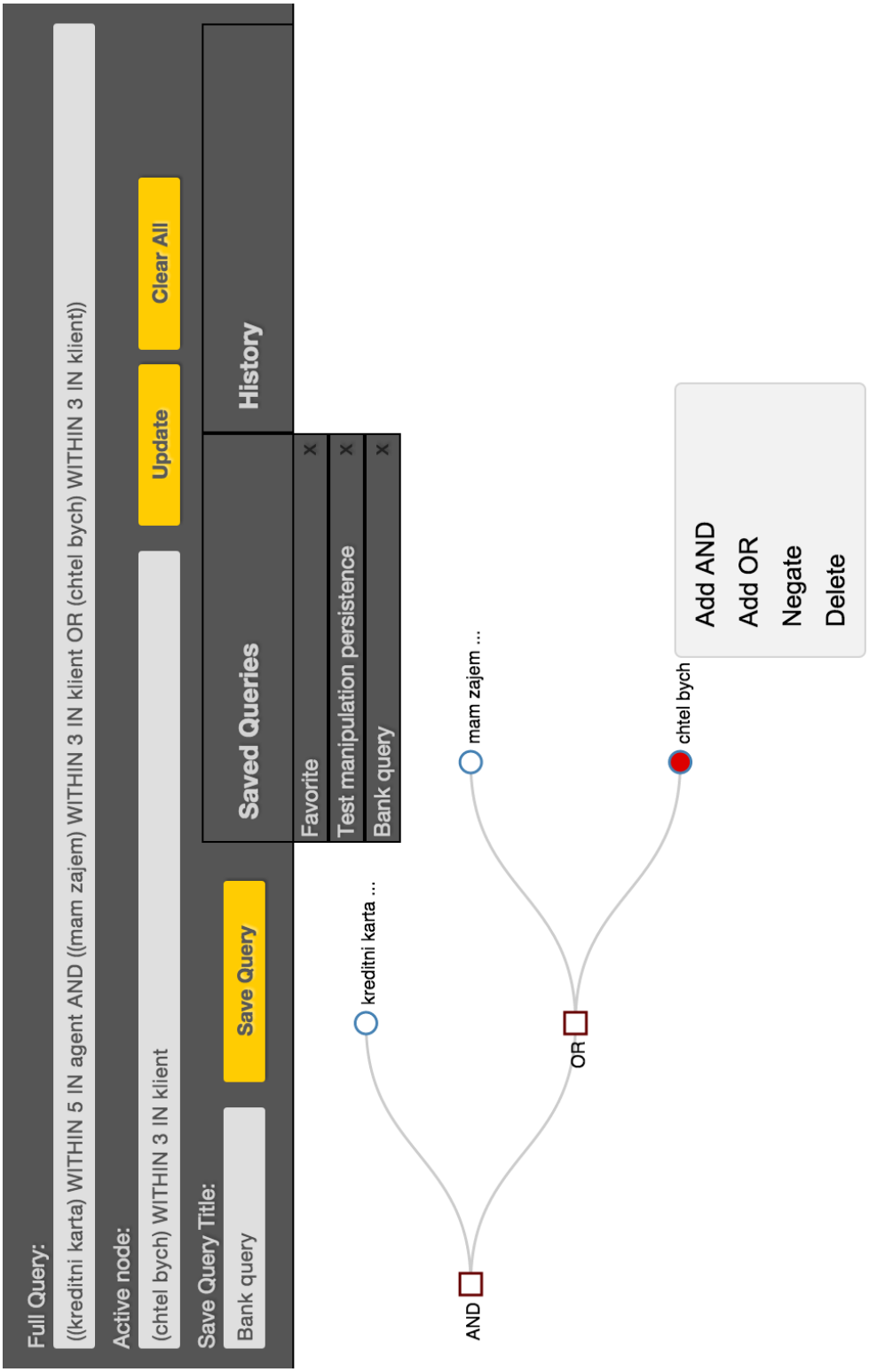

Figure 4.1: Seach Query Builder's GUI final version

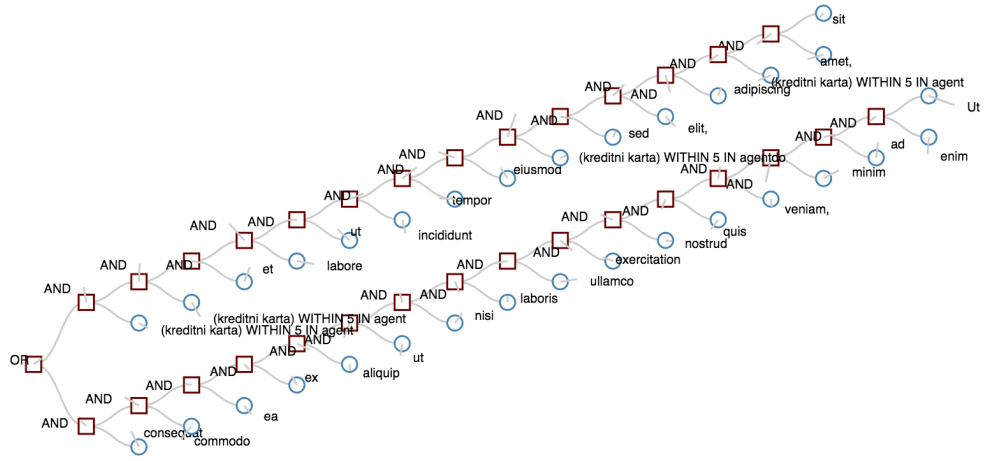

Figure 4.2: Force-based label placement

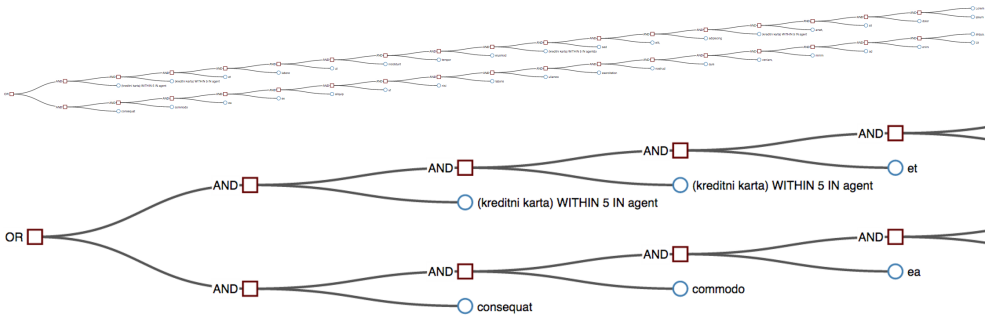

Figure 4.3: Final version of label placement. The overview of the query is at the top part of the figure. The zoomed in view is shown at the bottom part.

Chapter 5

Testing

To prove that a software does what it was designed for and to eliminate errors before its delivery to end users, it should be tested and the search query builder module is no exception. There are several types of tests and each of them focuses on a different aspect of the tested software.

The unit testing is the first type of testing I did. It verifies that small units of the module work as is expected of them. An important part of the query builder is the parser, so I focused mainly on that. The principle of testing the parser was simply to provide it with a query string and then check if it returned the expected parsed object. This proved to be very useful when developing the parser as it quickly discovered any changes in functionality.

After testing the individual parts, I needed to make sure the integration of the backend with the frontend worked. For this purpose, I wrote several integration tests that performed actions on the frontend that required the cooperation of backend. Very simple example of such a test is to programmatically save a query on the frontend and check that the returned object from the backend has an identification number that should be automatically generated by a database.

Another type of testing I had access to, was the user testing. The module was tested by the IBM experts and I also received a feedback from the IBM customers who the module was presented to and who would potentially use it in production environment. This is how the problem with label placement, discussed in chapter 4, arose. It also led to discovering several types of queries that were being parsed incorrectly. Later, I added those queries into unit tests to make sure the parsing errors do not repeat.

5.1 Technology

I used two additional tools for unit and integration testing purposes. Mocha is a JavaScript test framework that automates testing and provides tools for understandable error reporting [38]. I also used an assertion library Chai [39], because Mocha itself does not have any built-in assertion language. The assertion language is a way of defining what value the variables should be and what functions they should have.

Chapter 6

Adaptation for other platforms

Even though the solution was designed to be as platform independent as possible, some measures still need to be taken to integrate the prototype with a different ESP.

6.1 Parser

Every ESP has a different syntax for expressing often similar features. This means that a new grammar has to be created for the Jison parser generator. The newly created parser can be then loaded very simply with the use of RequireJS dependency loader.

6.2 Query building

After the query has been modified, it has to be reassembled back into the query string form. This is the opposite process to the parsing and also it is specific to every ESP. This process happens in the buildString method of the SQB package (see figure 3.3) and its output needs to be adapted to correspond to the target syntax.

6.3 Styling

The last step is to modify the looks of the GUI by creating and adding custom CSS to make the module go with the style of the ESP. This step is optional and it does not in any way alter the behaviour of the module.

Conclusion

The hypothesis for this thesis is that queries can be visualized. In the analysis, I have proved there are common patterns among the QLs of various ESPs. Later, I have designed and implemented a prototype, that utilizes these common patterns for visualization of the queries. The prototype also contains the management of queries via persistence of the favorite queries and history of the last performed queries. I have also proposed necessary steps for adapting the prototype for other ESPs, because it is bound to work only with IWCA's QL as it is. Thereby I have accomplished all the tasks set up in the assignment.

To make the implemented prototype ready for production use, I will continue the development process in cooperation with IBM. There are two areas I want to focus on. Firstly, I want to add the user management to the backend. It would ensure that users could not see each other's favorite queries, if they do not choose to. For security reasons, the user management is expected to depend on other professional system for managing user identities such as the Lightweight Directory Access Protocol (LDAP). Secondly, inspired by the jQuery Query Builder, I would like to develop ways for inputting specific fields such as dates and numbers, when building a query. This would potentially eliminate many syntax errors when building a query.

Bibliography

[1] Gartner 2014 Magic Quadrant Report. [Cited 2015-3-7]. Available from: http://www.coveo.com/en/resources/ebooks-white-papers/

[2] Gartner Magic Quadrant [online]. [Cited 2015-3-25]. Available from: http://www.gartner.com/technology/research/methodologies/research_mq.jsp

[3] Bennett, M. 20+ Differences Between Internet vs. Enterprise Search - And Why You Should Care [online]. 2008, [Cited 2015-2-25]. Available from: http://www.ideaeng.com/inet-enterprise-search-p1-0502

[4] White, M. *Enterprise Search*. Enhancing business performance, Oreilly & Associates Incorporated, first edition, 2012, ISBN 9781449330446.

[5] Page, L.; Brin, S.; Motwani, R.; et al. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999, previous number = SIDL-WP-1999-0120. Available from: http://ilpubs.stanford.edu:8090/422/

[6] Koren, J.; Zhang, Y.; Liu, X. Personalized Interactive Faceted Search. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-085-2, pp. 477–486, doi:10.1145/1367497.1367562. Available from: http://doi.acm.org/10.1145/1367497.1367562

[7] Gincel, R. Focusing Enterprise Search. *InfoWorld*, volume 26, no. 42, 2004: pp. 36 – 41, ISSN 01996649. Available from: http://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=14744937&site=ehost-live&scope=site

[8] Solr Query Syntax [online]. [Cited 2015-3-25]. Available from: http://wiki.apache.org/solr/SolrQuerySyntax

[9] Apache Lucene – Query Parser Syntax [online]. [Cited 2015-3-25]. Available from: http://lucene.apache.org/core/3_5_0/queryparsersyntax.html

[10] ElasticSearch reference: Queries [online]. [Cited 2015-3-25]. Available from: http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-queries.html

[11] Coveo Platform 7.0: Search Prefixes and Operators [online]. [Cited 2015-3-25]. Available from: http://onlinehelp.coveo.com/en/ces/7.0/user/search_prefix_and_operators.htm

[12] Google Search Appliance Documentation: Request Format [online]. [Cited 2015-3-25]. Available from: http://www.google.com/support/enterprise/static/gsa/docs/admin/72/gsa_doc_set/xml_reference/request_format.html

[13] HP Idol OnDemand: APIs [online]. [Cited 2015-3-25]. Available from: https://www.idolondemand.com/developer/apis

[14] Content Analytics with Enterprise Search 2.2.0: Search Syntax [online]. [Cited 2015-3-25]. Available from: http://www-01.ibm.com/support/knowledgecenter/SS5RWK_2.2.0/com.ibm.discovery.es.ap.doc/iiyspqysyntx.htm

[15] Building search queries in SharePoint 2013 [online]. [Cited 2015-3-25]. Available from: https://msdn.microsoft.com/en-us/library/office/jj163973.aspx

[16] Microsoft SQL Server Query Builder. [Cited 2015-3-25]. Available from: https://technet.microsoft.com/en-us/library/ms186906(v=sql.105).aspx

[17] DbVis Software Query Builder. [Cited 2015-3-25]. Available from: https://www.dbvis.com/features/tour/query-builder/

[18] SQLeo Visual Query Builder. [Cited 2015-3-25]. Available from: http://sourceforge.net/projects/sqleo/

[19] dbForge Query Builder for SQL Server. [Cited 2015-3-25]. Available from: https://www.devart.com/dbforge/sql/querybuilder/

[20] Easy Query Builder. [Cited 2015-3-25]. Available from: http://easyquerybuilder.com/

[21] Active Query Builder. [Cited 2015-3-25]. Available from: http://www.activequerybuilder.com/

[22] jQuery QueryBuilder. [Cited 2015-3-25]. Available from: http://mistic100.github.io/jQuery-QueryBuilder/

[23] Carter, Z. Jison. [Cited 2015-4-29]. Available from: http://zaach.github.io/jison/

[24] Majda, D. PEG.js [online]. [Cited 2015-4-29]. Available from: http://pegjs.org/

[25] Demaille, A.; Eggert, P. GNU Bison [online]. [Cited 2015-4-29]. Available from: https://www.gnu.org/software/bison/

[26] Bostock, M. Reingold–Tilford Tree [online]. [Cited 2015-5-1]. Available from: http://bl.ocks.org/mbostock/4339184

[27] D3.js [online]. [Cited 2015-4-30]. Available from: http://d3js.org/

[28] Gillespie, P. d3-context-menu [online]. [Cited 2015-5-2]. Available from: https://github.com/patorjk/d3-context-menu

[29] jQuery [online]. [Cited 2015-5-2]. Available from: https://jquery.com/

[30] basil.js [online]. [Cited 2015-5-2]. Available from: http://wisembly.github.io/basil.js/

[31] Git [online]. Available from: https://git-scm.com/

[32] Yeoman [online]. [Cited 2015-4-30]. Available from: http://yeoman.io/

[33] Dojo Toolkit [online]. [Cited 2015-4-30]. Available from: https://dojotoolkit.org/

[34] RequireJS: A JavaScript Module Loader [online]. [Cited 2015-4-30]. Available from: http://requirejs.org/

[35] Stefaner, M. Force-based label placement [online]. [Cited 2015-4-30]. Available from: http://bl.ocks.org/MoritzStefaner/1377729

[36] Williams, J. Solving D3 Label Placement with Constraint Relaxing [online]. [Cited 2015-4-30]. Available from: https://blog.safaribooksonline.com/2014/03/11/solving-d3-label-placement-constraint-relaxing/

[37] IBM Knowledge Center: Creating and deploying a plug-in to add custom widgets for user applications [online]. [Cited 2015-4-30]. Available from: http://www-01.ibm.com/support/knowledgecenter/SS5RWK_3.5.0/com.ibm.discovery.es.ap.doc/iiyspplugschviews.htm

[38] Mocha [online]. [Cited 2015-4-30]. Available from: http://mochajs.org/

[39] Chai Assertion Library [online]. [Cited 2015-4-30]. Available from: http://chaijs.com/

Appendix A

Programmer's manual

A.1 Frontend

The frontend project structure was generated with the use of Yeoman. The tools used for further development and their installation are described at `http://yeoman.io/learning/`. These are:

Bower.io (`http://bower.io/`) – a dependency manager for the frontend development

GruntJS (`http://gruntjs.com/`) – a task runner, that can automate various repetitive tasks like running a server, testing or building the plugin

NodeJS (`http://nodejs.org/`) – a JavaScript runtime

After installing these tools, it is necessary to run `npm install` and `bower install` commands from the root folder of the frontend. These commands will install all the remaining defined dependencies. Now, starting the web server and serving frontend files is done with the `grunt serve` command. The structure of the project follows the conventions of the Yeoman's generator called generator-webapp (`https://github.com/yeoman/generator-webapp`).
The JavaScript code follows the conventions described at `http://javascript.crockford.com/code.html`.

A.1.1 Testing

I used two tools for testing purposes. Both of them should be installed after running the `bower install` command. These are:

Mocha (`http://mochajs.org/`) – a testing framework

Chai (`http://chaijs.com/`) – an assertion library

The necessary documentation for writing further tests is available at their websites.

A.2 Backend

The backend is implemented in Java EE and requires a Java EE compliant application server for running. I used the IBM WebSphere Liberty Profile application server (`IBMWebSphereLibertyProfile`). I also used the IntelliJ IDEA development environment, which allows fast deployment to the application server, but any other should suffice. For configuration of the application server, consult the documentation. I include the configuration of my `server.xml` file for reference. It also contains the configuration of the Derby database.

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">
  <featureManager>
    <feature>jsp-2.2</feature>
    <feature>localConnector-1.0</feature>
    <feature>jaxrs-1.1</feature>
    <feature>servlet-3.0</feature>
    <feature>jdbc-4.0</feature>
    <feature>jndi-1.0</feature>
    <feature>jpa-2.0</feature>
  </featureManager>
  <httpEndpoint id="defaultHttpEndpoint" httpPort="9080"
    httpsPort="9443" />
  <applicationMonitor updateTrigger="mbean" />
  <jdbcDriver id="DerbyEmbedded" libraryRef="DerbyLib" />
  <library id="DerbyLib" filesetRef="DerbyFileset" />
  <fileset id="DerbyFileset"
    dir="${shared.resource.dir}/derby/lib" includes="derby.jar" />
  <dataSource id="jdbc/sqb" jndiName="jdbc/sqb"
    jdbcDriverRef="DerbyEmbedded">
    <properties.derby.embedded databaseName="sqbDerby"
      createDatabase="create" />
  </dataSource>
  <dataSource id="jdbc/sqbnonjta" jndiName="jdbc/sqbnonjta"
    jdbcDriverRef="DerbyEmbedded" transactional="false">
    <properties.derby.embedded databaseName="sqbDerby"
      createDatabase="create" />
  </dataSource>
  <application id="SearchQueryBuilder_war_exploded"
      location="{{ path to artifact }}"
```

```
      name="SearchQueryBuilder_war_exploded"
      type="war" context-root="/sqb" />
</server>
```

Appendix B

Acronyms

AJAJ asynchronous JavaScript and JSON.

AJAX asynchronous JavaScript and XML.

API application program interface.

CRUD create, read, update and delete.

CSS Cascading Style Sheets.

DOM Document Object Model.

ES enterprise search.

ESP enterprise search platform.

GPS Global Positioning System.

GSA Google Search Appliance.

GUI graphical user interface.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

IWCA IBM Watson Content Analytics.

Java EE Java Enterprise Edition.

JSON JavaScript Object Notation.

LDAP Lightweight Directory Access Protocol.

MVC Model-View-Controller.

QL query language.

REST Representational State Transfer.

SVG Scalable Vector Graphics.

URI uniform resource identifier.

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
  - `SearchQueryBuilder.war` ..... the backend in WAR format
- `src` ..... the directory of source codes
  - `searchQueryBuilder` ..... implementation sources
    - `backend` ..... backend implementation sources
    - `frontend` ..... frontend implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format