Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

User interface development for advanced searching in the MBDB

Vývoj uživatelského rozhraní pro pokročilé vyhledávání v projektu MBDB

Bachelor's Degree Project

Author:	**Kryštof Krejčí**
Supervisor:	**Ing. Jakub Klinkovský, Ph.D.**
Consultant:	**Dr.rer.nat. Emil Dandanell Agerschou**
Language advisor:	**Bc. Nathaniel Tobias Patton**
Academic year:	2023/2024

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Krejčí**	Jméno:	**Kryštof**	Osobní číslo:	**503183**

Fakulta/ústav: **Fakulta jaderná a fyzikálně inženýrská**

Zadávající katedra/ústav: **Katedra matematiky**

Studijní program: **Aplikovaná informatika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vývoj uživatelského rozhraní pro pokročilé vyhledávání v projektu MBDB

Název bakalářské práce anglicky:

User interface development for advanced searching in the MBDB project

Pokyny pro vypracování:

1. Seznamte se s projektem MBDB, jeho vizí a aktuálním stavem.
2. Nastudujte strukturu databáze, obsažené entity a roli metadat.
3. Rozšiřte projekt MBDB o počáteční implementaci "pokročilého vyhledávání", která umožňuje sestavit, editovat a vykonat dotazy popsané pomocí JSON, a také zobrazit výsledky hledání.
4. Dbejte na to, aby implementované funkce byly uživatelsky použitelné a otestované.

Seznam doporučené literatury:

[1] Wilkinson, M., Dumontier, M., Aalbersberg, I. et al. The FAIR Guiding Principles for scientific data management and stewardship. Sci Data 3, 160018 (2016). https://doi.org/10.1038/sdata.2016.18
[2] Atzeni, P., Bugiotti, F., Cabibbo, L., and Torlone, R. Data modeling in the NoSQL world. Computer Standards & Interfaces, 67, 103149 (2020). https://doi.org/10.1016/j.csi.2016.10.003
[3] Percival, H., and Gregory, B. Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. O'Reilly Media, Inc. (2020). ISBN: 978-1492052203
[4] Stránský, J., Williams, M., and Mas, C. Databases related to molecular biophysics 09-2022 [Data set]. Zenodo (2022). https://doi.org/10.5281/zenodo.7114168

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jakub Klinkovský, Ph.D.   katedra softwarového inženýrství   FJFI

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Dr.rer.nat. Emil Dandanell Agerschou   Biotechnologický ústav AV ČR

Datum zadání bakalářské práce: **31.10.2023**   Termín odevzdání bakalářské práce: **05.08.2024**

Platnost zadání bakalářské práce: **30.09.2025**

[illegible]	Masáková	[illegible]
Ing. Jakub Klinkovský, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Zuzana Masáková, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. Ing. Václav Čuba, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

28.11.2023	Krejčí
Datum převzetí zadání	Podpis studenta

Acknowledgment:
I would like to thank my supervisor, **Ing. Jakub Klinkovský, Ph.D.**, for his expert guidance and unwavering support throughout this project. Great guidance and insights from him were invaluable in shaping the direction and success of my work.

I would also like to express my gratitude to **Dr.rer.nat. Emil Dandanell Agerschou** for his practical advice and consultations. His guidance was crucial in implementing the project and ensuring the accuracy of the code through his thorough reviews.

Lastly, I would like to thank **Bc. Nathaniel Tobias Patton** for his assistance with the language, which helped in refining the final presentation of this thesis.

Author's declaration:
I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 5, 2024 Kryštof Krejčí

Název práce:

Vývoj uživatelského rozhraní pro pokročilé vyhledávání v projektu MBDB

Autor: Kryštof Krejčí

Studijní program: Aplikovaná informatika

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Jakub Klinkovský, Ph.D., katedra softwarového inženýrství FJFI

Konzultant: Dr.rer.nat. Emil Dandanell Agerschou, Biotechnologický ústav AV CR

Abstrakt: Tato práce popisuje vývoj pokročilého uživatelského rozhraní pro vyhledávání v Molecular Biophysics Database. Kromě toho se snaží kombinovat moderní webové technologie, jako jsou React a Semantic UI, společně s robustní backendovou podporou z Pythonu a Elasticsearch, aby zvýšila efektivitu získávání dat a zlepšila celkový uživatelský zážitek. Konkrétně se tato práce zaměřuje na implementaci dynamického tvůrce vyhledávacích dotazů, pomocí kterého mohou uživatelé snadno vytvářet složité vyhledávací dotazy. Poté je uživatelsky vytvořený vyhledávací dotaz odeslán na server. V tomto systému server zpracovává požadavek a poté vrací odpovídající výsledky vyhledávání. Systém také zahrnuje mnoho nezávislých funkcí: konverze YAML na JSON a specificky vyvinutý validační proces k zajištění integrity dat. Toto nové rozhraní významně zlepšuje vyhledávací schopnosti MBDB, poskytuje cenný nástroj pro výzkumníky v oblasti molekulární biofyziky a zlepšuje celkové řízení a dostupnost dat.

Klíčová slova: pokročilé vyhledávání, uživatelské rozhraní, molekulární biofyzika, MBDB, React, Elasticsearch, převod YAML do JSON, vyhledávání dat

Title:

User interface development for advanced searching in the MBDB

Author: Kryštof Krejčí

Abstract: This thesis describes the development of an advanced user interface for searching within the Molecular Biophysics Database. Moreover, it seeks to blend modern web technologies such as React and Semantic UI, combined with solid backend support from Python and Elasticsearch, to increase data retrieval efficiency and enhance the overall user experience. Specifically, this thesis focuses on implementing a dynamic search query builder, through which users can easily create complex search queries. Then the user-created search query is sent to the server. In this system, the server processes the request and then returns the appropriate search results. The system also includes many independent features: a YAML-to-JSON conversion system and a specifically developed validation process to guarantee data integrity. This new interface significantly enhances the search capabilities of MBDB, providing a valuable tool for researchers in molecular biophysics and improving overall data management and accessibility.

Keywords: advanced search, user interface, molecular biophysics, MBDB, React, Elasticsearch, YAML to JSON conversion, data retrieval

Contents

Chapter 1

Introduction

1.1 Challenges with Existing Database Search Methods

The ability to search in complex scientific databases effectively is critical for researchers to gather the information they need to advance their work. The amount and complexity of data are rapidly increasing, posing challenges when using traditional search methods. Researchers must sift through vast amounts of specialized data to find the exact information they seek. Moreover, the average researcher in the biological field is unlikely to have received training in SQL or similar query languages, making it even more challenging to perform advanced searches.

Limited Search Flexibility Most of today's traditional search methods depend on keyword matching which can be insufficient for more complex scientific queries. Researchers may be struggling with expressing their search criteria, which can lead to incomplete or unrelated data.

Complex Data Scientific databases like MBDB contain data with complex relationships. Keyword matching is not enough for these datasets. Effective use of metadata can drastically increase search efficiency.

User Interface and Experience The complexity of advanced search requires an intuitive user interface without a steep learning curve. A poorly created interface can discourage users from using this feature and cause them to miss out on valuable data, given that many researchers in the biological fields are not trained in complex database querying, a user-friendly interface becomes even more critical.

Performance With a growing amount of data in a database, ensuring quick search performance may become difficult. Researchers don't want to wait for slow responses during complex queries.

Better Data Discovery After solving the problem of basic keyword searching issues, researchers are able to find relevant datasets faster. This can reduce the time needed to find needed information, giving them more time for research and development.

Goals of the Thesis The primary goals of this thesis are to develop an advanced user interface for the MBDB, enhance the usability of the search functionalities, and ensure the system is accessible to researchers with varying levels of technical expertise. This involves integrating modern web technologies, ensuring robust backend support, and implementing user-friendly features for creating, saving, and loading complex search queries.

1.2 Significance of the MBDB in Biophysical Research

The MBDB is evidence of the vital role that data sharing and accessibility play in promoting scientific advancement. Furthermore, the searchable format of the database not only enhances the visibility of distinct research projects but also makes it possible to validate pre-existing models as well as create new ones through meta-analyses. In addition, the project encourages a more transparent and collaborative research environment by highlighting the value of open science and data-sharing guidelines. This is particularly important in the context of the so-called 'reproducibility crisis' in experimental sciences, where the ability to replicate results is crucial for scientific progress. This thesis explores scientific research challenges and the technical aspects of the project while developing new search features for MBDB. The new search tools in the MBDB can improve the speed of finding relevant data, which will accelerate the understanding and progress of molecular biophysics.

1.2.1 GitHub Repository

The development and implementation details of this advanced search feature can be found in the MBDB GitHub repository under the mbdb-search project. This repository contains all the code, documentation, and additional resources for the project. By providing public access to the repository, we encourage collaboration, transparency, and further development within the research community [1].

1.3 MOSBRI Integration

The MBDB is aligned with the objectives of the MOSBRI (Molecular-Scale Biophysics Research Infrastructure) initiative, which aims to develop a pilot database for biophysical data. The integration with MOSBRI allows MBDB to leverage shared resources and standards, enhancing data interoperability and collaboration within the biophysical research community. For more information, refer to the MOSBRI objectives [2].

Chapter 2

Introduction to the MBDB

The Molecular Biophysics Database (MBDB) is a vital tool for scientists, making new advances in how researchers save and share data. This project wants to push molecular biophysics ahead by putting high-quality, regulated biophysical data in one place for easy use. MBDB's core goal is to create one place that keeps all the records of all experiments safe and helps people use them in the future. It gathers a great number of experiments together in one place that users can search through, making MBDB a beacon of hope for people all over the world to team up, share data openly, and make science data available to everyone. It promises to facilitate easy access to data and make science experiments reproducible. This lays a strong base for discoveries and cross-disciplinary studies and goes past old limits. By bringing together what we know from the molecular biophysics community, MBDB significantly aids in our search to learn more about how molecules shape life and disease.

2.1 Objectives of the MBDB

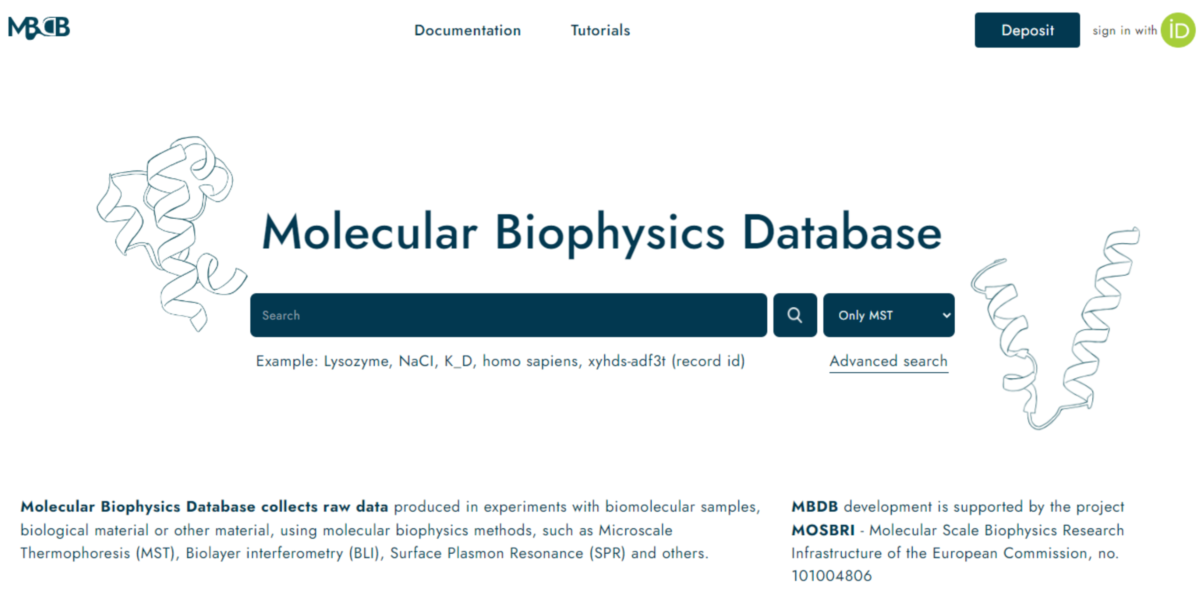
The primary goal of the MBDB is to create a comprehensive repository that collects biophysical data from all sorts of experiments, and data analyses. The repository aims to serve as a resource for researchers, students, and others by providing easy access to data from all around the world.

- Increase the efficiency of scientific research by enabling quick and reliable data retrieval.
- Assist interdisciplinary research by providing a platform that combines information from different biophysical sub-fields.
- Encourage the repurposing of current data to cut down on redundant data generation and stimulate creativity, for data-driven discoveries and establishing where existing gaps are.

2.2 Core Technology for Data Storage

At the technological heart of the MBDB is Invenio, which is an open-source framework designed for managing digital repositories which was chosen for its strong feature set, scalability, and adaptability. The capability of the framework when dealing with large datasets, complex data models, and various metadata standards makes it an ideal choice for the project. Invenio is also used for databases such as Zenodo or CERN Open Data [3]. Invenio helps MBDB in the following main aspects:

- **Modular Architecture**: Because of Invenio's modular architecture, the MBDB can be expanded and customized to satisfy the changing demands of the biophysics community.



Molecular Biophysics Database

Example: Lysozyme, NaCl, K_D, homo sapiens, xyhds-adf3t (record id)

Molecular Biophysics Database collects raw data produced in experiments with biomolecular samples, biological material or other material, using molecular biophysics methods, such as Microscale Thermophoresis (MST), Biolayer interferometry (BLI), Surface Plasmon Resonance (SPR) and others.

MBDB development is supported by the project **MOSBRI** - Molecular Scale Biophysics Research Infrastructure of the European Commission, no. 101004806

Figure 2.1: Main page of the MBDB [4].

- **Easy Data Model**: By adopting Invenio, MBDB creates a comprehensive data model that takes into account the unique characteristics of biophysical data, such as polymer structures and experimental conditions
- **Advanced Search Functionality**: Invenio has a powerful search engine, which is based on Elasticsearch that enables complex querying.

2.2.1 Key Components of Invenio

Invenio has the following components:

- **API**: Manages content negotiation, security, and schema validation using tools like Marshmallow and JSON Schema.
- **Records**: Utilizes JSON Schema for defining data structure, ensuring consistency and accuracy.
- **Search and Indexing**: Powered by Elasticsearch, it provides advanced search capabilities for efficient data querying.
- **Metadata Extraction**: Allows automatic extraction and management of metadata, crucial for effective data discovery.
- **UI**: User-friendly interface for search, deposit, detail views, and administration, facilitating easy interaction with the repository.

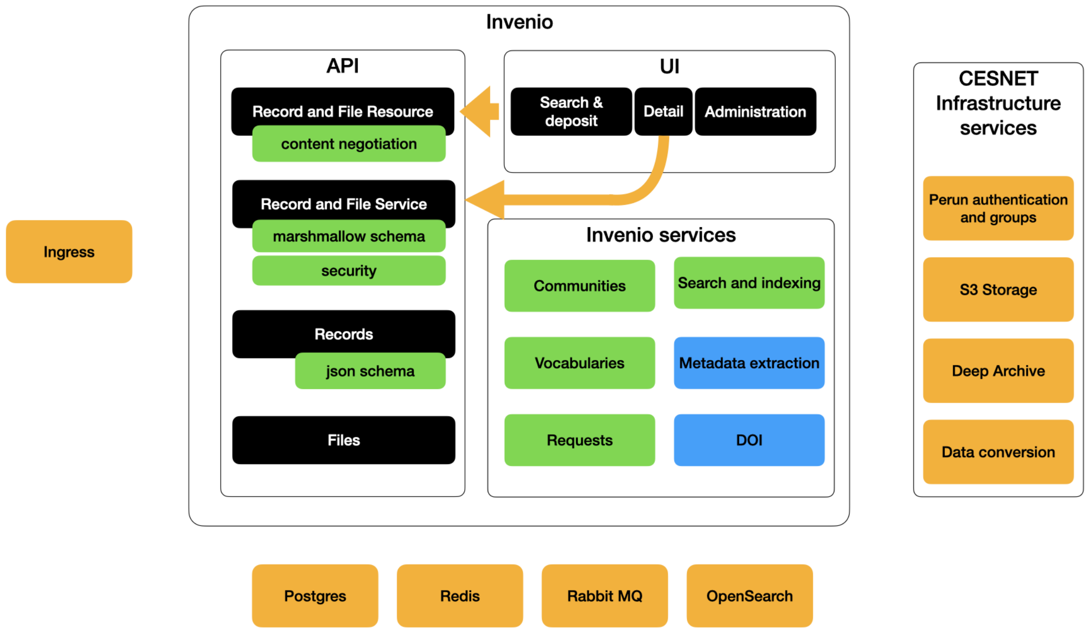


Figure 2.2: Invenio repository architecture[5].

2.3 Database Structure and Metadata

The main model of MBDB is a two-part structure, which differentiates between general parameters and method-specific parameters.

General Parameters

General parameters describe both the metadata of the record (*e.g.* authors), as well as biophysical properties expected to be present for all types of biophysical measurement (*e.g.* identities of measured species).

All records have the same fields in the general parameters section Independent of measurement technique, which means that required fields in general parameters are present (and hence searchable) in all records.

Method-Specific Parameters

The method-specific parameters include detailed information about how data was measured and how it was analyzed (*e.g.* measurement protocol). As this is inherently specific to the techniques used, the fields in method-specific parameters should only be expected to exist in records using the same technique.

2.3.1 Structure of the Model

Reusable Elements

Reusable elements are designed to be used across different data models, providing consistency and reducing redundancy in the data structure.

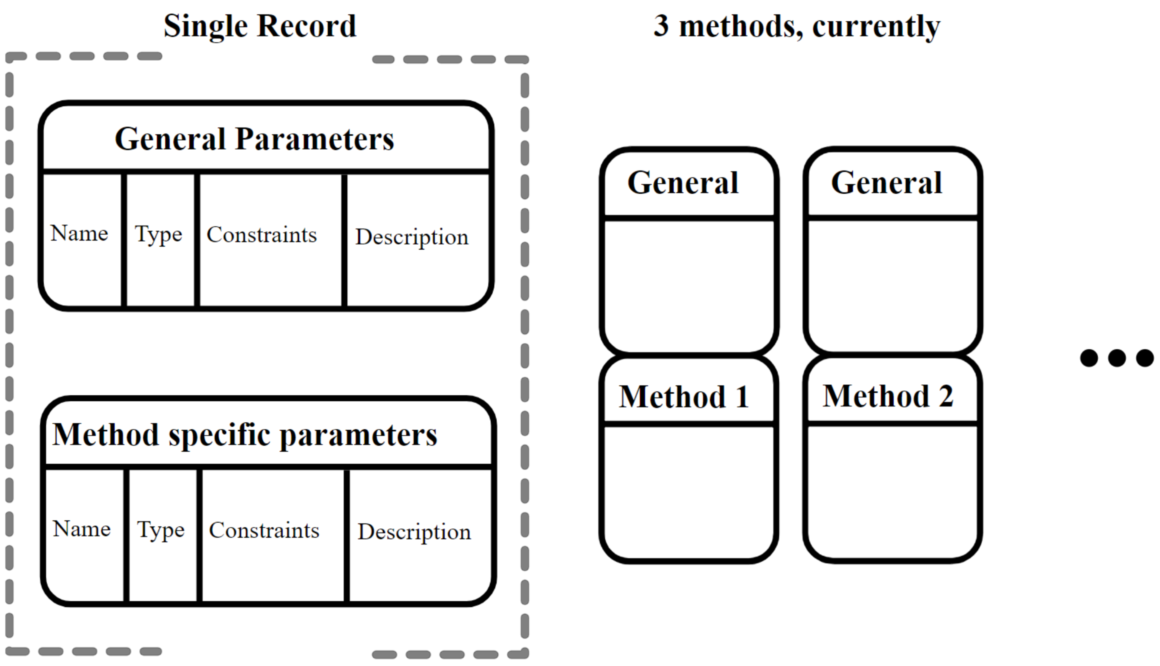


Figure 2.3: Two-part structured model, general parameters and method-specific parameters [7].

Polymorphic Elements

Polymorphic elements allow for flexible data modeling by supporting different data types and structures, which is crucial for accurately representing the diverse range of biophysical data. You can either look at it as an analog of class inheritance or perhaps more appropriately as an implementation of this schema pattern. The Polymorphic Pattern is used when documents have more similarities than they have differences, allowing them to be stored in a single collection for improved performance. This pattern is particularly useful when we want to access information from a single collection without the need for complex joins. For more details, refer to the MongoDB Polymorphic [8].

2.4 Tools Used in This Thesis

This section discusses the different tools that were used during the development of the Advanced Search project. Each subsection will focus on one particular tool or library that was chosen for the project, its purpose, and its application within the project.

2.4.1 React and Semantic UI

React is a JavaScript library that is used to build user interfaces, specifically those of single-page applications where data can be changed dynamically without necessarily reloading the whole page. Backed by React 18, the MBDB creates a user interface that responds promptly—including delivery of real-time information [9]. With React's component-based architecture, user interfaces are made in a modular way which means they can be managed more easily and updated independently without affecting other components within an application. The advantage of this approach is that it simplifies how we develop web applications as chunks that can just be moved around within an application which would have been otherwise not possible.

Semantic UI is a front-end development framework that aims to create beautiful, responsive layouts using human-friendly HTML. These components were integrated with React in the MBDB to enhance the user experience [10].

BUN is a modern JavaScript runtime that drives the UI in MBDB, with all the necessary tools and configurations necessary for a smooth process of designing and deploying software. Thus, BUN is made to save time and be effective as it lines up the stages of creating an application and the final product's operation. Using BUN, the MBDB gets faster production, faster code execution, and a wide range of tools to handle dependencies and create the application [11].

2.4.2 Python

Python, known for its simplicity and versatility, is extensively used in the MBDB for various backend processes. It is very heavily used language both within and outside of the scientific, making it easier to make collaborative development. Various scripts were developed with Python to convert data, connect to Elasticsearch, and carry out challenging searches. When dealing with large datasets, it is possible to carry out manipulations on them fast without any difficulties because Python has many libraries like pandas and NumPy among others that can be used for data processing efficiently [12].

2.4.3 Apache Lucene

Apache Lucene is an open-source search library developed by the Apache Software Foundation. It provides powerful search capabilities and forms the backbone of many search applications, including Elasticsearch. In the MBDB, Lucene enables fast and accurate retrieval of biophysical data through its robust query language and efficient search algorithms [13].

2.4.4 Elasticsearch

Elasticsearch is a strong search and analytics engine that is employed in indexing and querying large datasets. By using Apache Lucene, it offers quick and precise search results. In the MBDB, Elasticsearch handles indexing and querying of biophysical data. Due to its scalability and distributed architecture, the engine can handle very large quantities of data both efficiently and quickly. Elasticsearch in the MBDB is suitable due to its advanced search capabilities, real-time queries, and powerful query language, allowing consumers to easily obtain the data they require [14].

2.4.5 Luqum

Luqum is a Python library that aids in the generation and analysis of sophisticated search inquiries. During the execution of MBDB, Luqum serves as a tool that helps to convert user input into organized queries that Elasticsearch can process efficiently. Lucene queries, on which Elasticsearch's query language is anchored, can be created by Luqum. It is used to make sure that the MBDB performs its search queries and gives back proper results to users [15].

2.4.6 JSON

JSON, which is short for JavaScript Object Notation, is a lightweight data interchange format that's comprehensible by humans and easy for machines to process and generate. Because of its simple syntax based on key-value pairs, JSON finds a lot of practical usage in web applications when it comes to exchanging data between them [16].

Within the MBDB, JSON is utilized for storage and data transmission, guaranteeing that secure user interface-generated formats can be analyzed by backend systems seamlessly.

2.4.7 YAML

YAML, which stands for "YAML Ain't Markup Language" is a standard for human-readable data serialization commonly used in configuration and data exchange. This language with easy syntax is often used for complex data description and is both easy to read and write so many people use it [17]. YAML and JSON are highly related and since YAML (1.2), JSON is equivalent to a subset of YAML's flow style syntax without comment.

The MBDB uses YAML files to define data models and configuration which are later transformed into JSON format for the system to continue the process.

Chapter 3

Advanced Search Frontend Development

3.1 Introduction

In this section we will cover how the Advanced Search module was designed and developed and look at how it is applied by the MBDB system. Most importantly, we will see how complex search queries can be generated or modified at high speed, with precision in results, and efficiently through the platform. It includes working on big datasets through a combination of YAML mapping and React-powered UI. First, we will explore the primary components of the software. This includes how queries are created by the web application and how querying interpreters process queries from back-end algorithms while transforming YAML into searchable JSON objects. This system's various parts have been engineered to work in perfect harmony ensuring that anyone using it receives nothing but the best experience possible.

3.2 Workflow

The main objective of this section is to provide a general description of the Advance Search processes. This section discusses what each process does, how it interacts with others, what it receives, and what it forwards to the next. At the beginning, the search method is defined in a YAML file. There are currently four methods defined in MBDB: `MST.yaml`, `BLI.yaml`, `SPR.yaml`, and `ITC.yaml`. It contains the MBDB database's searchable items. The project now uses the first one, MST.yaml, however, it is simple to exchange it with the other two methods to get different results.

3.2.1 Yaml Mapping

The YAML tree structure has to be converted into individual objects with their descriptions so that the user can view each one as a searchable option. This is the point at which my `convert_yaml_to_json.py` is useful. It accomplishes this by recursively traversing and extracting the YAML. Following the extraction, the names may be ambiguous. Because there are multiple objects with the same ending names but different paths to them. Therefore, we must improve their user-recognition. The goal is to make each name unique. The `prettier_names.py` process is the one that performs this task. It traverses and comes up with the best names by going over the objects several times. Once this is finished, it is prepared for display.

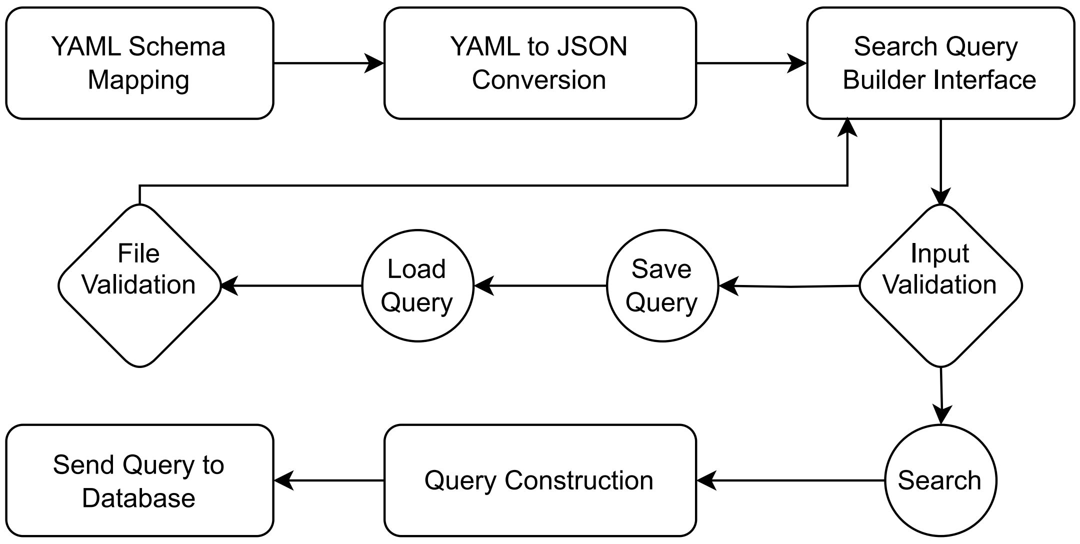


Figure 3.1: The diagram shows how are each component and process connected.

3.2.2 React-Based Application

The React-based web application is the most crucial component of the entire process. This is where the user assembles and refines the search query. After the user is satisfied, the query has to pass additional validations to ensure it is correctly constructed. The user can either search the assembled query or store it for later use after the query has been verified.

3.2.3 Save and Load

If the user chooses to save the query, the website's content is converted into JSON format, shown in the console, and an input request for the file name appears.

By using the file as a backup, the query does not need to be assembled from scratch each time. More experienced users may change or combine many searches. (Combine two or more JSON files)

After that, users have the option to load the file into the web application. Where it needs to pass another validation process. Therefore, the user is unable to upload invalid data or inject malicious data into the server. The UI is then reconstructed based on the uploaded data so that it may be altered and searched for.

3.2.4 Search

Continuing with the search functionality, we can now search using two separate methods. The first one reads the lines and converts them into a string, which is then sent to the API as (q=...). The second one, more complex sends the data to the server utilizing the same JSON structure that was used to save the query, but instead of sending sting, it sends (json=...).

3.2.5 JSON to Elasticsearch

When the server receives this JSON data, it first transforms it into a Luqum query and then into an Elasticsearch query, which is then sent as a request to the database to retrieve the data. Lastly, the data that the database returned is displayed on the testing version of the MBDB application available at `https://mbdb.test.du.cesnet.cz/`[4]. This testing version allows users to interact with the application and verify its functionality before it is deployed in a production environment.

3.3 User Interface

The whole interface of the query builder sits behind the Advanced Search button on the MBDB web page. This means users can easily access advanced search functionalities with a single click. After clicking the button, the query builder interface is revealed. It is a line-by-line input form with logical operators between every two lines except the first one, as a single line cannot logically compare itself. Then the user searches and chooses what object they are looking for. Then corresponding value field is selected. Then *to-value* input can be generated with the plus ("+") button. On every line are left and right brackets, activated by a click. Also if the user figures out that they made a mistake in the process of creating multiple lines, they can remove individual lines with the Remove button. Under the query are four buttons: Add Field, Search, Download/Copy, and Upload. The input is made of lines where the user can put in what you're searching for. On each line, you can pick from `AND`, `NOT`, or `OR` operators, except on the first line. Then you choose what object the user is searching for and type in the specific thing they want. If it's a date or number, they can also look for a range, for example from January to March. Each line lets the user use brackets to be more specific. After setting up their search, pressing the search button will get results. The user can also save their searches, copy them, or load them again. It is designed to be user-friendly.

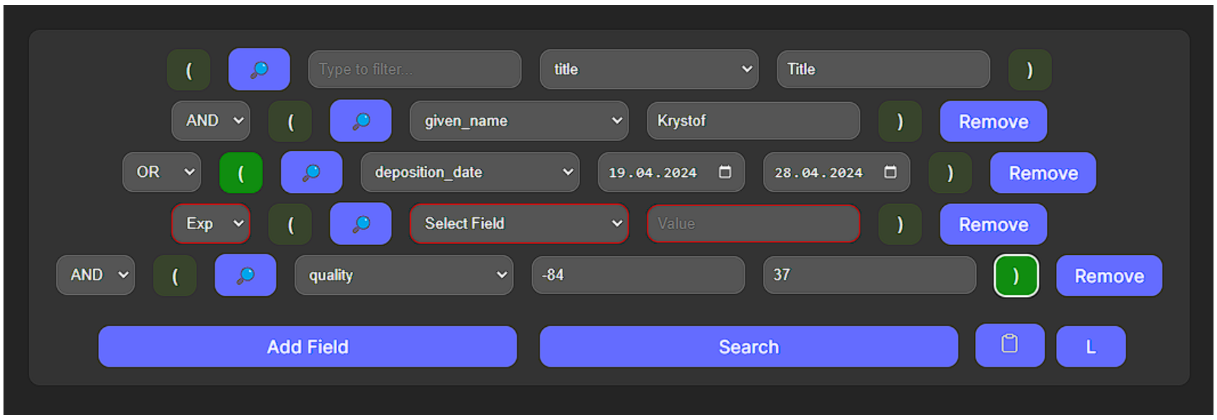


Figure 3.2: The image shows all the functionality of the query builder

3.3.1 Operators

As mentioned earlier, the operators are the first thing that should be filled in each line. The operator in the first line is not present because at least two query lines are needed. Three operators are available to use: `AND`, `OR`, and `NOT`. Operators `AND` and `OR` work as expected. The `NOT` operator works as an `AND` operator but negates the value for the line or bracket segment. This implies that `AND` and `NOT` operators can be on the same logical level. But `AND` and `OR` cannot. More about this is the validation chapter [4].

3.3.2 Searchable Fields Filter

There is a great amount (450-500 depending on the specific model) of searchable items in the database for users to search for. For this exact reason, there is a search field that will narrow down the *searchable objects*. Clicking on the magnifying glass on each line will cause it to expand into an input field. After inserting the value, it will narrow the search for the *searchable objects*.

3.3.3 Searchable Fields

This input determines the parameters for the search as well as the kind of input that will be displayed for the *from-value* and *to-value*. It is recommended to use the filter to reduce the number of available options.

3.3.4 Value Input

Depending on what type of object is being searched for, this input field will change after it has been selected. At the moment, there are three primary input types: dates, numbers, and strings. When using the string input, the user is not restricted in any way to what can be entered. Any positive and negative numbers can be entered into the number type input. Finally, the user is prompted to enter a legitimate date because the date type input will appear as a date picker. If present, it is not possible to input larger or smaller numbers than what is allowed for the number input. This can be considered a form of input field validation.

3.3.5 Range Value Input

The *to-value* is hidden beneath the "+" button and only appears for date and number inputs. After clicking, an input field matching the value type is created. It functions just like the *from-value* input. There is one connection between the two. *From-value* input functions as a *from* value, while *to-value* input functions as a *to*. When *from* is greater than *to*, the two values are swapped to keep consistency in the searches.

3.3.6 Brackets

Each line has one set of brackets, the left after the operator and the right one at the end of the line. They have two states, dark green to show they are disabled and bright green for active ones. Brackets are also validated with multiple measures that are discussed in the validation chapter [4].

3.3.7 Buttons

Beneath the query input area, four essential buttons are displayed, providing users with a range of functionalities to enhance their search experience:

- **Add Field**: This button helps users add more fields to the search query in a dynamic way. Every additional click introduces another line for field input and allows users to further clarify or expand their search criteria.
- **Search**: Centered on the interface is where you will find this button that carries out an act of searching the search. As soon as you have set the conditions that you want, all it takes is just one single click on this same button for it to return with results having met any specific criteria given.

- **Download/Copy**: For those who work with the same queries, copy and save button should be more useful. It takes time to enter lengthy queries into the inputs. With the help of this feature, users can save their query as a JSON file or view it in the console.
- **Load**: The user can load the query into the web page from the JSON file without any issues. Errors may arise if they decide to change the file. The user is notified when the query loads partially or not at all.

3.3.8 Alerts

The user receives alerts when a validation process detects errors in the query or during the JSON loading process on the website. For the convenience of the user, all errors are displayed in a single alert. See the validation section for more information on alerts.

3.4 Metadata Model Parsing Algorithm

For the YAML configuration to be accessible and searchable in my application. The YAML first needs to be converted into JSON format that describes every searchable object. This process is divided into two main parts.

1. **YAML to JSON Conversion** (`convert_yaml_to_json.py`)
   The `convert_yaml_to_json.py` script reads the YAML file, then parses its content, and finally constructs a JSON array where is every element represented as a field.
2. **Normalization of Field Names** (`prettier_names.py`)
   Many fields now have ambiguous names that can be confusing for the user. The `prettier_names.py` takes care of this problem and ensures a unique name for each field.

3.4.1 YAML to JSON Conversion

The MBDB relies heavily on a YAML to JSON conversion. This is because it changes YAML's tree-like structure into JSON format which has become popular. This helps a lot in handling, querying, and displaying data within the system efficiently. I developed a Python script for this specific purpose, which reads the YAML file and then generates an equivalent JSON output after processing it.

Core functionality The script's primary functions include:

1. **Creating field items** The `create_field_item()` function constructs dictionaries for each field item, including properties like `pretty_name`, `field_path`, `type`, and `description`.
2. **Building JSON output** The `build_json_output()` function recursively processes the YAML schema, handling properties and polymorphic types, and constructing the JSON output. It includes:
    - **Field types**: The algorithm supports fields such as `string`, `number`, and `date`, determined based on the YAML type definitions.
    - **Additional properties**: Fields can include `minimum`, `maximum`, and `poly_type` properties if specified in the YAML.

Field creation in each object Each object in the JSON output can include multiple fields taken from the YAML schema, such as:

- **pretty_name**: Basic name for the object.
- **field_path**: The hierarchical path to the field within the schema.
- **type**: The data type of the field, such as `string` or `number`.
- **description**: An optional description of the field.
- **minimum** and **maximum**: Optional properties for numerical fields.
- **poly_type**: An optional property for polymorphic types.

This structured approach ensures that the YAML-defined data structures are accurately converted into a JSON format, handling all the data that can be queried within the MBDB system.

3.4.2 Normalization of Field Names

In the MBDB's JSON Pretty Name Standardization process, field names of the JSON data are made unique and human-readable. This entails reading a JSON file, identifying and resolving duplicate field names, and then changing them in a way that is standard.

Core functionality The script performs the following key tasks:

1. **Reading and writing JSON files**
    - **read_from_file(file_name)**: Reads JSON content from a specified file.
    - **write_to_file(file_name, data)**: Writes the processed JSON data to a specified file.
2. **Formatting names format_name(name)**: Standardizes field names by replacing underscores with spaces and capitalizing the words.
3. **Updating pretty names update_pretty_name()**: Updates the `pretty_name` field of each JSON object to ensure uniqueness and readability.
4. **Identifying and resolving duplicate names**
    - **find_duplicate_names()**: Identifies duplicate `pretty_name` fields and their paths.
    - **remove_if_equal()**: Removes common elements in paths to create minimal unique names.
    - **update_duplicate_pretty_names()**: Finds and updates duplicate pretty names within the JSON data.

Field creation in each object Each object in the JSON output includes:

- **pretty_name**: A user-friendly name for the field.
- **field_path**: The hierarchical path to the field within the schema.

This approach ensures that the JSON-defined data structures are standardized and unique, facilitating efficient data handling and querying within the MBDB system.

3.5 Backend Query Handling

This chapter covers the backend mechanisms for handling user-generated queries within the MBDB. The process involves converting user input from JSON to a query language that Elasticsearch can understand, ensuring efficient and precise data retrieval. The primary components responsible for this functionality are implemented `luqum_convertor.py` files. This files work to interpret, construct, and apply complex search queries.

3.5.1 Query Interpretation and Construction

General Workflow

The backend query handling begins with user-defined criteria in JSON format. These criteria are parsed and transformed into a structured query using the Luqum library, which is specifically designed for building and analyzing search queries. The constructed Luqum tree is then converted into an Elasticsearch query, which is executed to retrieve the relevant data.

Key Components in `luqum_convertor.py`

Method: `construct_luqum_tree` The `construct_luqum_tree` method is responsible for transforming JSON criteria into a Luqum tree. This involves:

- Parsing JSON criteria to identify fields, operators, and bracket operations.
- Constructing Luqum tree nodes for words, phrases, and ranges.
- Applying logical operators (`AND`, `OR`, `NOT`) to combine the fields into a coherent search tree.
- Handling nested queries using bracket operations to maintain the correct logical grouping.

Method: `apply` The `apply` method takes the constructed Luqum tree and converts it into an Elasticsearch query. It supports schema analysis to optimize the query structure and applies the query to the search object. Additionally, it handles aggregations and post-filters if specified in the JSON criteria.

The `luqum_convertor.py` file complements `search_options.py` by providing a standalone function for constructing Luqum trees from JSON criteria.

3.5.2 Integration in MBDB

The integration of these components within the MBDB allows for advanced search capabilities. Users can define complex search criteria through the web interface, which are then processed and executed by the backend, ensuring accurate and efficient data retrieval.

Key Components in `search_options.py`

The `search_options.py` file is a critical part of the query handling mechanism. It includes the `JsonQueryParamInterpreter` class, which contains methods to construct and apply search queries.

Class: `JsonQueryParamInterpreter` This class inherits from `ParamInterpreter` and includes methods for constructing Luqum trees from JSON criteria and applying these queries to the search object. ParamInterpreter is an Invenio native class, making this a seamless integration.

3.5.3 Detailed Example

To illustrate the functionality, consider the following example of a user-defined search query in JSON format:

```
[
  {
    "field": "metadata.general_parameters.record_information.title",
    "value": "Research Paper"
  },
  {
    "operator": "and"
  },
  {
    "field": "metadata.general_parameters.record_information.deposition_date",
    "value": {
      "from": "2024-01-01",
      "to": "2024-12-31"
    }
  }
]
```

This query searches for records with the title "Research Paper" and deposition dates within the year 2024. The backend components parse this JSON, construct a Luqum tree, and convert it into an Elasticsearch query to retrieve the relevant records.

3.5.4 Conclusion

The backend query handling in MBDB is designed to efficiently process complex search queries defined by users. By leveraging the Luqum library and Elasticsearch, the system ensures precise and fast data retrieval, enhancing the overall usability and functionality of the MBDB.

3.6 Code Functionality

This chapter provides an overview of the main code functionality within the MBDB application. The application is primarily built using React, a JavaScript library for building user interfaces, and it leverages several vital functions to handle the core operations of the application. This section will outline the key components and their main functionalities.

3.6.1 Main Code Overview

The application's main code is organized into several key files, each responsible for specific functionalities. The two primary files discussed in this chapter are `App.jsx` and `SearchCriteria.jsx`. These files contain the core logic for initializing the project, handling search criteria, and managing user interactions.

3.6.2 `App.jsx`

The `App.jsx` file is the entry point of the application. It initializes the project and manages the overall state of the application. The main functions within this file are:

- **App**: This function is responsible for the initialization of the project. It sets up the initial state and prepares the application for user interactions.
- **handleSearch** and **handleCriteriaChange**: These functions handle the criteria changes. `handleSearch` processes the search criteria and updates the state accordingly, while `handleCriteriaChange` manages the updates to individual search criteria inputs.
- **handleSearchClick**: This function triggers the search operation. It gathers the current search criteria, formats them, and sends the search request to the backend.
- **addSearchCriteria**: This function adds a new line of search criteria. It dynamically updates the state to include additional search criteria inputs.
- **handleJsonData** and **handleLoadJson**: These functions manage JSON data. `handleJsonData` processes the JSON data received from the server or user input, and `handleLoadJson` handles loading JSON data into the application's state.

`App.jsx` Code Excerpt

The following code is an illustrative overview of the main structure of the `App.jsx` file.

```jsx
import React, { useState } from 'react';
import SearchCriteria from './SearchCriteria';
import { handleSearch, handleCriteriaChange, handleSearchClick, addSearchCriteria,
    handleJsonData, handleLoadJson } from './functions';

const App = () => {
  const [criteria, setCriteria] = useState([]);

  return (
    <div>
      <h1>MBDB Search</h1>
      <SearchCriteria
        criteria={criteria}
        onCriteriaChange={handleCriteriaChange}
        onAddCriteria={addSearchCriteria}
      />
      <button onClick={handleSearchClick}>Search</button>
      <input type="file" onChange={handleLoadJson} />
    </div>
  );
};

export default App;
```

Listing 3.1: App Component

3.6.3 SearchCriteria.jsx

The `SearchCriteria.jsx` file defines the search criteria component. This component manages the individual search criteria and their associated functions. The main functionalities include:

SearchCriteria is the main component that defines the structure and behavior of the search criteria. It handles the rendering of input fields, dropdowns, and other UI elements related to search criteria. Additionally, it manages the functions for adding, updating, and removing search criteria.

`SearchCriteria.jsx` Code Excerpt

The following code is an illustrative overview of the main structure of the `SearchCriteria.jsx` file.

```
import React from 'react';

const SearchCriteria = ({ criteria, onCriteriaChange, onAddCriteria }) => {
  return (
    <div>
      {criteria.map((crit, index) => (
        <div key={index}>
          <input
            type="text"
            value={crit.value}
            onChange={(e) => onCriteriaChange(index, e.target.value)}
          />
          <button onClick={() => onAddCriteria()}>Add</button>
        </div>
      ))}
    </div>
  );
};

export default SearchCriteria;
```

Listing 3.2: SearchCriteria Component

3.6.4 Conclusion

This chapter has provided an overview of the main code functionality within the MBDB application, focusing on the `App.jsx` and `SearchCriteria.jsx` files. The key functions within these files are essential for initializing the project, managing search criteria, and handling user interactions.

Chapter 4

Testing and Validation

Every project needs to be tested and validated. The aim of testing and validation is to reduce unexpected behavior that might happen when a user uses the application in particular or edge scenarios. The developer additionally utilizes the validation as a precaution.

4.1 Input Form Validation

Validation in this project occurs in two stages. The first stage ensures that the user has entered a proper query into the inputs, checking for issues that can be created by the user.

1. Operator, field, *from-value*, and *to-value* inputs cannot remain empty.
2. The values are switched if the *from-value* is not smaller than the *to-value*.
3. Nothing other than numbers may be entered into the number input field.
4. Nothing other than the date can be entered into the input.
5. Minimal and maximal value control: No smaller or larger number than allowed may be entered into the input.
6. Operator validation. The `OR` operator cannot be on the same level as `NOT` and `AND`.
7. Make sure there are the same number of left and right brackets in the query.
8. Verifies that the brackets are in the right location and the query has a valid nesting structure.

The second type of validation serves as a guard of the JSON that the user tries to load into the interface. The JSON guard ensures consistency when loading. The user is informed by an alert when something goes wrong.

1. When the JSON doesn't have the correct structure, it is automatically rejected.
2. Look for missing fields
3. Detect missing operators
4. Swaps *from-value* and *to-value* in the same way as mentioned before.
5. Ensures that the values are in the valid range between minimal and maximal.

All input fields are mandatory, requiring the user to provide a value. When the input is left empty, it is highlighted with a red border, so the user doesn't have to go one by one and search for the empty inputs.

Consistency is vital in searching. After inserting from and to values in the wrong order, such that the from value is larger. In this case the inputs exchange values.

Brackets are validated on two different fronts. The first one only checked if there is the same number of left and right brackets. Sometimes it is sufficient for it to be valid. But when you take into account the nesting of brackets, another problem arises. It is necessary to make sure it is also satisfied.

4.2 Operator Validation

The main concern here is expression structure logic, namely the incompatibility of the `AND` and `NOT` operators at the same level as the `OR` operator. This is because combining these operators without using the appropriate hierarchical architecture results in unclear expression interpretation and logical problems. The creation of a string that shows the logical structure of a query depends on the function `generateQueryStructureString`. It achieves this by assembling a template that includes grouping symbols and logical operators from search criteria. Here, it is more important to illustrate the logic of the query than its actual meaning.

Then it is up to the `isValidExpression` function to make sure the logical expression follows predetermined rules. There are several steps involved in this:

1. **Clean:** Extra spaces are first removed to keep the expression uniform for analysis.
2. **Replace:** The algorithm identifies sub-parts which are enclosed in parentheses. Then they are replaced with a placeholder Y. It makes the query easier to understand. The nested parts are marked for further evaluation.
3. **Evaluate:** The main objective is to determine whether the simplified expression follows the given rules. It makes certain that the operators `AND` and `NOT` do not exist at the same level as `OR`.

Until the logical consistency of the entire structure is confirmed, these procedures are applied recursively to each component of the expression.

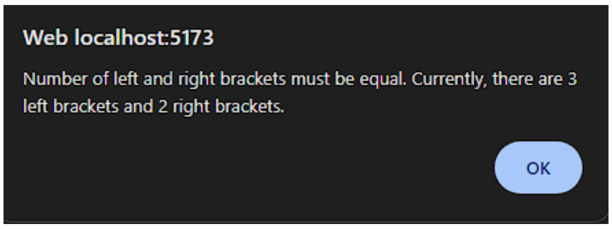
4.2.1 Example

The process of validating the logical structure of expressions is similar to validating a formal grammar, like L-systems. This ensures that the structure adheres to specific rules, maintaining logical consistency.

- **X** represents a single line in the expression.
- **Y** is a placeholder for a nested part of the expression.

Consider the expression: (X or (X not X) or X) and X. The `isValidExpression` function goes through the steps like this:

1. **Clean:** Remove extra spaces (Xor(Xnot$X$or$X$)or$X$)and$X$
2. **Replace:** (Xor(Xnot$X$or$X$)or$X$) replaced with a placeholder Y. Then we have $Y$and$X$


Web localhost:5173

Number of left and right brackets must be equal. Currently, there are 3 left brackets and 2 right brackets.

Figure 4.1: Search error alert

3. **Evaluate:** $Y$and$X$ is just one operator that implies validity.
4. The following steps are:
5. Xor(Xnot$X$or$X$)or$X$ $\rightarrow$ Xor$Y$or$X$ $\rightarrow$ valid
6. Xnot$X$or$X$ $\rightarrow$ has both `NOT` and `OR` that is by the rules invalid query. In this step it ends and False is returned.

4.3 Alerts

Users can make mistakes at every step of the search process, from editing the JSON to creating the search query on the website. When they make a mistake, we need to convey the message of what are they doing wrong, or at least what is happening. This is the reason why alerts have been implemented. These alerts will be displayed when the search query is wrong or incomplete. The other way how to trigger the alerts is when the validation of the loaded JSON is invalid. In the alert is a detailed description of all the problems. For each action, there is only one alert so the user does not need to click "OK" for every mistake he makes. some problems can be in the alert window multiple times because they can occur on multiple lines. In this way, the user knows it occurred multiple times on different lines.

4.4 Python Tests

The system contains Python tests that are created using the Python library pytest. Currently, tests are in the `test_luqum_convertor.py` file that can be run with the command `pytest test_luqum_convertor.py`. The test focuses on making sure that the Search feature operates correctly. We make sure the searches are consistent when changes are made and the format is not broken when we send our query to Elasticsearch.

4.4.1 Single Fields

These tests focus on the construction of single fields in the Luqum search tree. Every test involves a JSON input that asserts that the output of the `construct_luqum_tree` function matches the expected Luqum tree structure. This part consists of 11 tests and each one tests one case, for example, full-text value, negative value, number value, Unicode value, range value, and wildcard range (using *).

4.4.2 Operators

This section has only two tests, one makes sure the `OR` operator works, and correctly other one tests the `AND` operator.

Conclusion

In this thesis, I have developed an advanced user interface for the MBDB. The built-in functionality in that interface allows for improved access to and use of advanced search technology, which enables easy building and running of complex queries. Users were not able to take advantage of the possibility that such powerful search facilities of the MBDB had opened up. It solves a number of problems related to searching voluminous, scientific databases. With a friendly user interface and intuitiveness, complex search query formulation is reduced to basic steps. Users are now able to define search parameters, apply logical operators, and manage nested queries in a clean and responsive interface powered by React and Semantic UI. This is particularly valuable for those researchers who have the task of sifting through a great number of special data to find particular information.

The UI supports saving the queries into JSON files and reloading them when need be, without necessarily starting from scratch. In essence, this feature improves user experience by handling complex search criteria to be reused. At the core of this development is that the frontend user interface links with the backend, which ensures any query formation and fetching of results are carried out without errors using Elasticsearch. It does so by translating the user input into structured JSON queries, which are later converted to Elasticsearch queries with the help of the Luqum library. This will make the search powerful and efficient, enabling the system to process intricate queries and give the results refined in time.

Also, other features of the system provide processes such as YAML-to-JSON with robust validation, which creates data integrity and consistency. The mechanism of validation will guarantee that both the user-generated queries and the JSON files being loaded are properly formatted and error-free. This is a two-fold process for maintaining the integrity of the system and in not encountering errors because of incorrect or poorly entered data. Such features are essential not only to keep the search results accurate but also for the database to remain a valuable resource for researchers in molecular biophysics.

I learned a lot of technologies from this project, namely JavaScript, React, BUN, and Semantic UI. Moreover, I used to apply most of the algorithms known to me while developing independent components and integrating them into larger systems. This is sure to help in the next job—either another independent project or working with some people on a bigger framework.

In summary, this project is intended to be a significant contribution to the MBDB, ensuring that users can search for and retrieve data more effectively. It is also an example showing that the user-friendly design of scientific databases gives more accessibility to advanced technologies.

Bibliography

[1] MBDB GitHub Repository. *MBDB-Search: User Interface Development for Advanced Searching in MBDB*. Available at: `https://github.com/Molecular-Biophysics-Database/mbdb-search`. Description: The repository contains the source code and documentation for the advanced search user interface developed for the MBDB project.

[2] MOSBRI (MoIecular-Scale Biophysics Research Infrastructure). *MOSBRI Objectives and Goals*. Available at: `https://www.mosbri.eu/`. Accessed on: [20.7.2024]. Description: The initiative aims to develop a pilot database for biophysical data, enhancing data interoperability and collaboration within the biophysical research community.

[3] Invenio Software. *Invenio: Open Source Framework for Managing Digital Repositories*. Available at: `https://inveniosoftware.org/`. Accessed on: [15.6.2024]. Description: Invenio provides a robust framework for building and managing digital repositories, used by institutions such as CERN.

[4] MBDB Testing Version. *Testing Platform for Molecular Biophysics Database*. Available at: `https://mbdb.test.du.cesnet.cz/`. Accessed on: [1.8.2024]. Description: A testing platform for the MBDB project, allowing users to interact with and verify the functionality of the database.

[5] Invenio Architecture. *Detailed Architecture of Invenio Repository Framework*. Available at: `https://narodni-repozitar.github.io/developer-docs/docs/technology/invenio/architecture/`. Accessed on: [15.6.2024]. Description: Provides an in-depth look at the modular architecture of Invenio, highlighting its scalability and adaptability for large datasets.

[6] MBDB Documentation. *Introduction to the Data Model*. Molecular Biophysics Database. Available at: `https://molecular-biophysics-database.github.io/mbdb-docs/datamodel/intro`. Accessed on: [15.6.2024]. Description: An overview of the data model used in MBDB, explaining the structure and organization of biophysical data.

[7] MBDB Two-part Structure Data Model Image. *Visual Representation of the Two-part Structure Data Model in MBDB*. Available at: `https://github.com/Molecular-Biophysics-Database/mbdb-docs/tree/main/public/static/img`. Accessed on: [15.6.2024]. Description: An image illustrating the two-part data model used in MBDB, showing the separation of general and method-specific parameters.

[8] Ken W. Alger, Daniel Coupal. *Building with Patterns: The Polymorphic Pattern in MongoDB*. Available at: `https://www.mongodb.com/developer/products/mongodb/polymorphic-pattern/`. Accessed on: [14.7.2024]. Description: This article explains the polymorphic pattern in MongoDB, used for handling documents with similar but not identical structures, improving performance and query efficiency.

[9] React. *React: A JavaScript Library for Building User Interfaces*. Developed by Facebook. Available at: `https://react.dev/`. Accessed on: [8.7.2024]. Description: Detailed documentation and user guide for React, a popular JavaScript library for building modern user interfaces.

[10] Semantic UI. *Semantic UI: A Front-end Development Framework*. Available at: `https://semantic-ui.com`. Accessed on: [8.7.2024]. Description: Comprehensive documentation for Semantic UI, which provides responsive and customizable front-end components.

[11] Bun. *Bun: A Modern JavaScript Runtime*. Available at: `https://bun.sh/`. Accessed on: [8.7.2024]. Description: Documentation and guides for Bun, a JavaScript runtime designed for efficient and fast development.

[12] Python. *Python: An Interpreted, High-level Programming Language*. Developed by the Python Software Foundation. Available at: `https://www.python.org/`. Accessed on: [8.7.2024]. Description: Python's official site, offering extensive documentation, tutorials, and resources for developers.

[13] Apache Lucene. *Apache Lucene: A High-performance, Full-featured Text Search Engine Library*. Developed by the Apache Software Foundation. Available at: `https://lucene.apache.org/`. Accessed on: [8.7.2024]. Description: Detailed documentation and user guide for Lucene, which powers the search functionality in many applications, including Elasticsearch.

[14] Elasticsearch. *Elasticsearch: A Distributed, RESTful Search and Analytics Engine*. Developed by Elastic NV. Available at: `https://www.elastic.co/`. Accessed on: [8.7.2024]. Description: Comprehensive documentation for Elasticsearch, a powerful search engine based on Lucene.

[15] Luqum. *Luqum: A Python Library for Lucene Query Manipulation*. Available at: `https://luqum.readthedocs.io/en/latest/about.html`. Accessed on: [8.7.2024]. Description: Documentation and usage guide for Luqum, which helps in generating and analyzing complex search queries in Lucene.

[16] JSON. *JavaScript Object Notation (JSON) Data Format*. Available at: `https://www.json.org/json-en.html`. Accessed on: [8.7.2024]. Description: Official site for JSON, providing specifications and examples for this widely used data format.

[17] YAML. *YAML Ain't Markup Language (YAML) Data Format*. Available at: `https://yaml.org/`. Accessed on: [8.7.2024]. Description: Official site for YAML, offering specifications and documentation for this human-readable data serialization standard.