



Assignment of bachelor's thesis

Title:	Smart search module for LearnShell
Student:	Matěj Karpíšek
Supervisor:	Ing. Jakub Žitný
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

LearnShell is a modular system for managing and performing exams with programming assignments in scripting languages, especially Shell. LearnShell currently offers only basic functionality for searching among students, solutions, assignments, and exams.

1. Analyze the current infrastructure architecture and propose a new "search" module that will speed up search queries between frontend, backend, and database.
2. Configure the search backend and implement the communication with the frontend.
3. Implement smart search in the frontend UI (full-text search, tags, modifiers, filters).
4. Make sure you document your code and cover it with tests.



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Bachelor's thesis

Smart search module for LearnShell

Matěj Karpíšek

Department of Software Engineering
Supervisor: Ing. Jakub Žitný

May 12, 2021

Acknowledgements

I would like to thank my supervisor Ing. Jakub Žitný for leading my thesis and his valuable advice. And thanks to my family for their support during COVID times which made many things more challenging.

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 accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 12, 2021

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Matěj Karpíšek. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Karpíšek, Matěj. *Smart search module for LearnShell*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

LearnShell je webovou aplikací pro automatickou opravu shellovských úloh s chybějící funkcionalitou pro vyhledávání. Tato práce se zabývá analýzou, návrhem, implementací a testováním nového chytrého vyhledávacího modulu pro zmíněný systém. Cíle bylo úspěšně dosaženo. Modul — který byl postaven na open-source vyhledávači Elasticsearch — podporuje fulltextové a inkrementální vyhledávání v reálném čase, boolovské operátory a několik typů dotazů.

Klíčová slova LearnShell, chytrý vyhledávací modul, webová služba, fulltextové vyhledávání, Elasticsearch

Abstract

LearnShell is a web application for the automatic correction of shell assignments with the missing searching functionality. This thesis deals with the analysis, design, implementation, and testing of a new smart searching module for the system. The goal has been successfully achieved. The module — which has been built upon an open-source search engine Elasticsearch — supports full-text, real-time, incremental search, bool operators, and several types of queries.

Keywords LearnShell, smart search module, web service, full-text search, Elasticsearch

Contents

Introduction	1
Goals	2
1 LearnShell	3
1.1 Tech stack	3
1.2 Architecture	9
1.3 Data model	10
1.4 Searching functionality	11
2 Analysis and design	13
2.1 Requirements	13
2.2 Full-text search	15
2.3 Data storage	16
2.4 Elasticsearch details	18
2.5 Design	21
3 Realization	25
3.1 Back end	25
3.2 Front end	31
4 Testing	33
4.1 Unit testing	33
4.2 Python unit tests	33
4.3 TypeScript unit tests	34
4.4 React component unit tests	35
Conclusion	37
Bibliography	39

A	Acronyms	41
B	Contents of enclosed CD	43
C	Figures	45
D	Result examples	49
E	User documentation	55
	E.1 SmartSearch - User documentation	56

List of Figures

1.1	Docker diagram [11]	9
1.2	LearnShell architecture diagram (Created using [15])	10
2.1	Indexing documents and resulting index (Source: [17])	15
2.2	The search module data model (Created using [15])	21
2.3	The search module architecture (Created using [15])	22
C.1	Difference between SPA and traditional web page [7]	46
C.2	LearnShell database diagram (Created in PyCharm)	47
D.1	A basic query example	50
D.2	A tooltip example for submission script that does not fit (same works for assignment descriptions)	51
D.3	An example of specific field query and <i>and</i> operator	52
D.4	An example of quoted query	53
D.5	An example of <i>not</i> operator and <i>and</i> operator	54

List of Tables

2.1	Comparison of Elasticsearch and RDBMS terminology	18
-----	---	----

List of Listings

1	Docker file example	8
2	An example of explicit mapping	19
3	GraphQL API definition example	22
4	Elasticsearch and Kibana docker compose configuration	26
5	Submission index declaration	28
6	QueryBuilder class	29
7	GraphQL API declaration example	30
8	GraphQL client example	31
9	Python unit test example	34
10	TypeScript unit test example	35
11	React component unit test example	35

Introduction

LearnShell is used to support the teaching of the shell at FIT CTU in Prague. It is a web application that has been firstly developed by one of the students and since then has been improved many times. It provides the functionality of assignments, exams, and their automated correction. One of its current shortcomings is that teachers cannot efficiently search among the many documents¹ LearnShell comes with.

This thesis deals with the mentioned problem and comes up with a smart search module that will improve the searching experience for users of the system. It follows the common software engineering approach: analysis, design, realization, and testing.

The first chapter is about the analysis of the current state of LearnShell. It mentions its architecture, data model, and all technologies that will influence the development of the search module.

The second chapter is about the analysis and design of the search module. It deals with the tools that will be used during the development and the reasons for their choice.

The third chapter deals with the realization of the results of the previous chapter. And describes how the tools were used to achieve the working solution.

The last chapter mentions unit testing and shows some of the examples of unit tests that were used during the development.

¹users, assignments, exams, submissions, corrections

Goals

The point of this thesis is to achieve the following goals:

- First of all, it is necessary to understand the system that will be modified, so the first goal is to analyze the infrastructure of LearnShell. The result will be the knowledge of its data model, architecture, the current searching functionality, and the technologies it is built on.
- Secondly, based upon the system owner requirements, the functional and non-functional requirements will be specified for the search module, which with the previous analysis of the infrastructure, will be the foundation for the following analysis, design, and implementation of the back-end² part of the module.
- Thirdly, using the API of the back-end part, the front-end³ part of the search module will be implemented.
- During the development, the code will be properly documented and tested.
- Finally, the achieved results will be summarized, evaluated and possible extensions of the module will be suggested.

²The part of a web application that takes care of business logic and data storage

³The part of web application user communicates with

LearnShell

1.1 Tech stack

In this section, the technologies (libraries, frameworks, software...) of LearnShell will be described. Some of them have a long history and will be mentioned only briefly, so more focus can be put on the newer ones.

1.1.1 PostgreSQL

To store its data persistently and securely, LearnShell uses PostgreSQL. It is an object-relational database management system whose origins date to 1986. Traditional relation database stores its data in tables where each column value is atomic. However, PostgreSQL comes with additional object-oriented features which, for example, allow to store complex data types such as arrays and JSON⁴. [1]

1.1.2 Django

Most of the web applications are built upon a framework and LearnShell is no exception. Django is a Python framework for web database-driven applications which takes care of most of the hassle of web development. It has a long history as it was created in 2003, which provides reliability behind the whole framework. Django puts emphasis on the ease of creating applications, reusability, security, scalability, and the principle of DRY⁵. [2]

Django leverages one fundamental principle called ORM⁶. As mentioned in the previous section, all data in a relational database are stored in tables. This leads to a problem since the data being worked within Python have an object-oriented structure. An object-relational mapper is a library that

⁴JavaScript Object Notation

⁵Don't repeat yourself

⁶Object-relational mapping

takes care of this problem by automatically transferring the data stored in a relational database for us. In Django, the mapper can be used by writing a model class that represents a single table in a relational database and also provides the API for CRUD⁷ operations. [3]

1.1.3 Web API

Web application programming interfaces enable web applications to expose their functionality to the outer world. REST⁸ and GraphQL are two different ways to build such interfaces and LearnShell uses both.

1.1.3.1 REST

Representational state transfer is a software architectural style for distributed systems that was developed by Roy Fielding in 2000. It must follow several guiding principles as explained in [4].

The key feature of REST is that all data is abstracted using resources. A resource represents an exposed object of the system and is uniquely identified. Even though REST is protocol agnostic, it is most commonly used with HTTP⁹ where each resource corresponds to a single web address and the CRUD operation is specified by the HTTP method. [4]

1.1.3.2 GraphQL

GraphQL is a query language for APIs and a runtime to handle those queries developed by Facebook and released as open-source in 2015. It tries to solve some of the common problems of REST APIs but comes with its own pitfalls. [5]

The key difference compared to REST is that the API is described in terms of data types and fields, not endpoints. All queries are sent to a single endpoint and using the GraphQL query language, describe the data we want to receive. A runtime then handles the query, ensures we ask for the types and fields that are described, and otherwise provides helpful errors. [5]

What makes GraphQL great is that you get only the data you ask for, avoiding under-fetching or over-fetching. You can get all the data in a single request, avoiding overloading the backend. And since GraphQL is strongly typed, you get the data in a predictable, secure way. [6]

Regarding the disadvantages. Firstly, queries can quickly become complex and hard to manage without the usage of proper patterns and tools. For example, queries requesting multiple different types with many nested fields. Secondly, all the queries must be parsed and executed by the GraphQL

⁷create, read, update, delete

⁸Representational state transfer

⁹Hypertext Transfer Protocol

runtime which results in a slight overhead. Lastly, there are problems with caching. Since GraphQL does not use different endpoints such as REST, it is not possible to implement HTTP level cache for requests. [6]

1.1.4 React, Next.js and Typescript

LearnShells frontend is built upon the concept of SPA¹⁰. Traditionally, whenever a new page is requested, the whole page is returned from the server and then the browser reloads for the changes to take effect. Single-page applications change this behavior and using the AJAX¹¹ technology, request only the data they need and then dynamically rewrite the page. [7] This leads to a more efficient, responsive, and desktop-like experience. The whole process can be seen in Figure C.1.

1.1.4.1 React

As mentioned in the previous section, single-page applications use AJAX technology. JavaScript changes the page dynamically by manipulating the DOM¹² which is a tree data structure representing the whole page. React is a JavaScript library to make such manipulations easier.

To achieve higher abstraction and declarative API, React comes with the concept of the Virtual DOM. React does not immediately manipulate the real DOM, but instead stores an ideal, or “virtual”, DOM representation in memory, which is then synced with the real one in a process called reconciliation. This enables us to avoid common JavaScript tasks such as attribute manipulation or manual DOM updating. Instead, it is declaratively told to React in what state the page is supposed to be in and it takes care of the rest. [8]

React applications are built out of components. Components are the building blocks of the page in React world, they manage their own state, encapsulate rendering logic and can be composed to create more complex ones. Since the drawing logic of a web page is specified with HTML¹³, React came with JSX which is a syntax extension to JavaScript. It enables us to write HTML-like code in JavaScript which is then transpiled¹⁴. [8]

React leverages several principles from the functional paradigm which must be followed when developing a component:

- Component’s state must be immutable. If we want to change it, we must create a new one and assign it using React’s API. The reason behind this is that React uses reference comparison to decide whether a component

¹⁰Single-page application

¹¹Asynchronous JavaScript and XML

¹²Document object model

¹³HyperText Markup Language

¹⁴Compiled from JavaScript source code with extensions to valid JavaScript source code

has changed and must be re-rendered. This makes the process extremely fast compared to checking every single field of the state object.

- Components can receive arbitrary data as a parameter called props. Those data are read-only and must not be modified which makes components pure functions.
- Composing components leads to a component hierarchy. In such a hierarchy, data must always flow one way only, from the parent components to the child ones. The only way for a child component to communicate with the parent is by a callback passed by props.

All of this is necessary to make React work properly. On top of that, it makes components easier to manage, test, and predict their behavior.

1.1.4.2 Next.js

Single-page applications are great, but they come with several problems. Since the web page is dynamically rewritten, the browser routing system does not support such behavior. Search engine crawlers usually do not execute JavaScript, which means they cannot see the dynamically generated content. The JavaScript code can become large, bloated with extensions, and include browser unsupported features. All of those problems and much more, are solved by React framework — Next.js. [9]

Next.js solves the routing problem by having its own file-system-based routing system. It works in a way that the project includes a specific directory *pages* where each file or directory corresponds to a page route automatically. Files contain a React component representing a page with the given route. [9]

To achieve better performance and SEO¹⁵ the framework makes use of a concept called pre-rendering. A page's HTML is generated in advance, instead of having it all done by JavaScript in the browser. [9]

Next.js also uses a bundler WebPack and a compiler Babel which requires zero configuration to work. They take care of the last-mentioned problem. [9]

1.1.4.3 Typescript

Next.js supports implicit TypeScript integration. TypeScript is a programming language developed by Microsoft and released in 2012. As its name suggests, it extends JavaScript with optional static typing. TypeScript is syntactically a superset of JavaScript, so JavaScript source code is valid TypeScript code. [10] The reason behind the language is that a lot of bugs can be caught ahead of time during compilation using static typing. Furthermore, IDEs¹⁶ can provide much better code completion so the documentation does

¹⁵Search engine optimization

¹⁶Integrated development environments

not have to be used as much. On the other hand, a little more code must be written.

1.1.5 Docker

The whole project, except a single service, which is not Docker compatible yet, is built upon Docker. Docker is OS-level virtualization technology and product released as open-source in 2013. Using Docker, the software is packaged into containers and distributed with its needed environment to be easily run on any Docker running platform. It started with already existing concepts in the Linux world, such as cgroups and namespaces, but came up with a single user-friendly solution. What makes Docker exceptional compared to virtual machines, even though they both offer resource isolation benefits, is the performance overhead. All containers share a single host kernel making it a lightweight solution, whereas virtual machines come up with their own kernel and hardware virtualization. [11]

1.1.5.1 Architecture

All containers are run on a client-server application called Docker Engine. According to [12] the engine consists of three major components:

Docker daemon

Like every daemon, it is a process that runs in the background and handles commands sent by clients. Its main purpose is to manage various Docker objects such as images, containers, networks, and volumes.

Docker client

It's a terminal or GUI application that's used whenever a user wants to make a Docker operation. All commands are delegated and handled by the Docker daemon

REST API

Works as a communication bridge between the client and the daemon.

1.1.5.2 Images

Images are multi-layer read-only templates for containers. Templates are in a way similar to the concept of classes and objects in object-oriented programming. Whenever we want to create and run a container, we need to have its image, to begin with. Multi-layer structure makes creating new images easier by creating new ones built upon existing ones. The fact that layers are read-only makes reusing layers across different images possible, avoiding duplication and saving a significant amount of memory in the end. [12]

A new image can be created upon a dockerfile which is a text file with Docker-specific instructions. All the instructions are executed in a sequential

manner and a new image is created based on them. We can see an example in Listing 1.

```
FROM ubuntu:20.04
CMD echo "Hello, World!"
```

Listing 1: Docker file example

As I've already mentioned, images are multi-layer. Therefore, how are all the different layers merged in the end to be used by a single running container? Docker came up with a solution based upon UnionFS. UnionFS allows merging files across different file systems called branches into a single virtual file system where contents of directories sharing the same paths will be all seen in a single logical directory. Name collisions are resolved by assigning different precedence to each branch. [13]

By leveraging the mentioned virtual file system, containers can be worked with as expected and in an efficient way.

1.1.5.3 Containers

The most fundamental concept in the world of Docker virtualization. Containers are built upon images and are the final working unit that packages all the code with its required environment. They can be run or stopped. They can form virtual networks. They can share the host file system. Among many other things. [11]

Making use of the multi-layer principle of images, containers are simply created by adding a new writable layer to the image. We can create several containers out of a single image with each having its own unique writable layer. [11]

By default, containers are stateless which means that all the data is lost after restarting the container. Containers usually connect to an external database service to store data persistently. Fortunately, Docker has its own solution called Docker volumes which make specified container directories persistent. [11]

Communication among containers is possible by another Docker object called network. Whenever containers share the same user-defined network, they are accessible through the container's name specified while creating. This works because of an internal domain name system that translates container names to their private IP addresses. [11]

1.1.5.4 Registry

Registry is what comes up naturally and makes Docker so pleasant to work with. It's a public repository where images can be stored and easily accessed.

Docker came up with its own registry Docker Hub but also unofficial ones exist. [12]

1.1.5.5 Compose

According to Docker documentation [11]: “*Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML¹⁷ file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.*”

Docker Compose is what makes working with multi-container applications much easier. Each service is a Docker image with additional configurations specified such as Docker volumes, environmental variables, and port mapping between Docker network and host network. Everything in one place.

1.1.5.6 Summary

All of the mentioned can be seen in action in Figure 1.1.

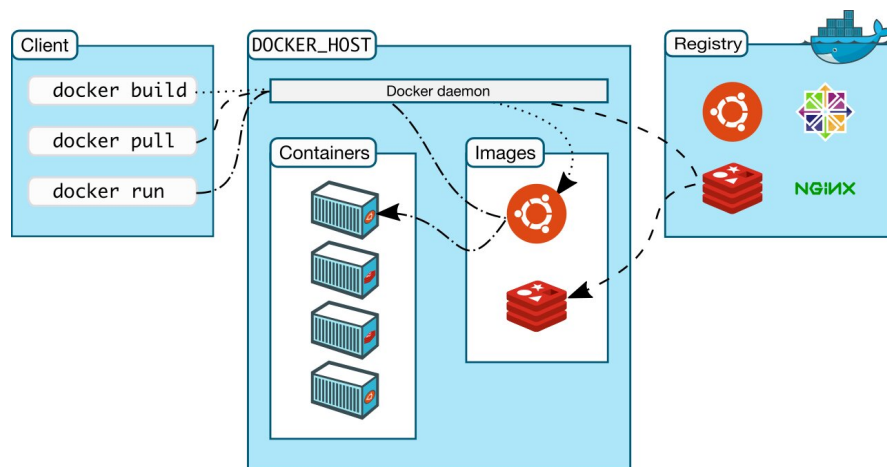


Figure 1.1: Docker diagram [11]

1.2 Architecture

LearnShell is a web application that follows a microservice architecture, which means that the application is divided into a set of loosely coupled services. Each of the services is easily maintainable, testable, and independently deployable. The communication between them is realized using REST or GraphQL API¹⁸. [14]

¹⁷A human-readable data-serialization language

¹⁸Application programming interface

The simplified version of the architecture is shown in Figure 1.2. Each box represents a service. The arrows connect communicating services. We can see that the frontend is a fully decoupled service that communicates with the backend via GraphQL API. This pattern is common for SPA web applications which request only the necessary data using JavaScript and then dynamically rewrite the current page.

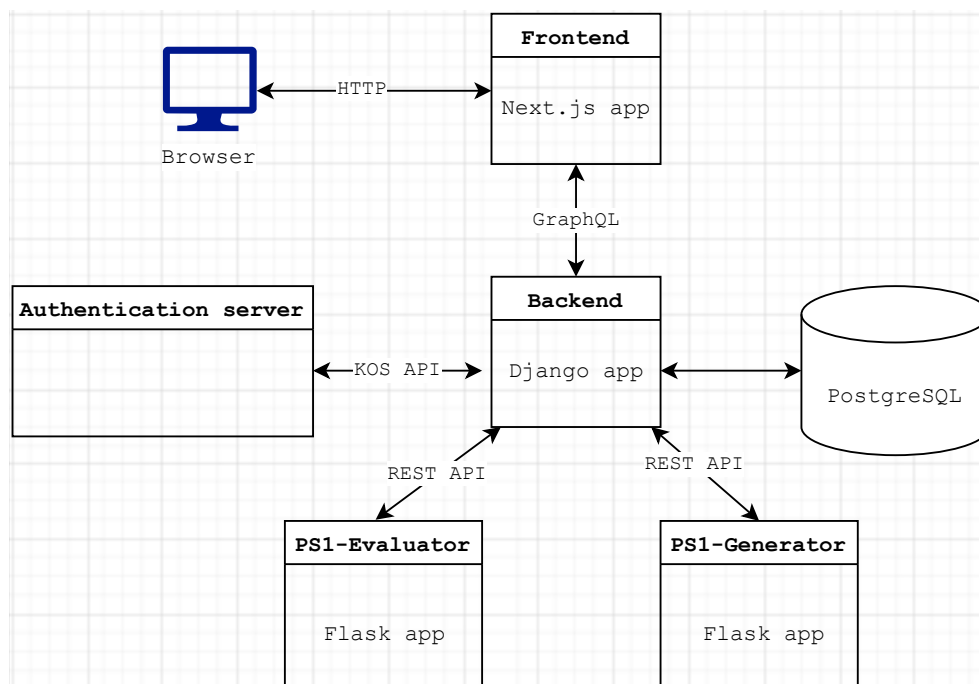


Figure 1.2: LearnShell architecture diagram (Created using [15])

1.3 Data model

As mentioned previously, LearnShell uses PostgreSQL RDBMS¹⁹. We can see in Figure C.2 that the database diagram is quite complicated, so only the significant entities are explained.

User

User represents a user of the system such as a student or a teacher. And is primarily used for authentication and authorization purposes.

Assignment

Assignments are a bit complicated since they are dynamically generated

¹⁹Relational database management system

using an external service for each student. *Assignmenttemplate* contains data about which generator and evaluator service is used for a given assignment. *Assignment* contains data for the services. *Generatedassignment* is generated by the generator service using the two mentioned entities for each student.

Exam

Assignments can be standalone or we can use exams to group several assignments. *Examtemplate* is used to specify to which course the exam belongs and which assignments it contains. *Assignmentexamtemplate* represents many-to-many relation between *Assignment* and *Examtemplate*. *Exam* represents an actual exam to be written by the specified teacher at a given time. *Studentwritesexam* is created when a student starts to write the exam.

Submission

Submission is created whenever a student submits a solution to some assignment. The data are base64²⁰ encoded. It is then asynchronously passed to an evaluator service to be corrected.

Correction

Correction is asynchronously added when the evaluator service completes the correction. Contains the result for given *Submission*.

1.4 Searching functionality

LearnShell currently does not have implemented any specific searching module in the frontend. Basic searching can be done manually by clicking through the details of entities listed in the frontend. This way is highly inefficient and unlisted entities are not even searchable.

In the backend, GraphQL API provides filtering by any field for all entities. On top of that, number fields support range queries and string fields support prefix, substring, suffix, and regex²¹ queries. The usability of this functionality will be elaborated in the upcoming chapter where a new search module and changes to the system will be designed.

²⁰Encodes binary to ASCII characters

²¹String matching given regular expression

Analysis and design

2.1 Requirements

A prerequisite for the success of a software project is a precise specification of the requirements which must be satisfied by the target product. A good requirement must be specific, measurable, achievable, and realistic.

2.1.1 Functional requirements

Functional requirements are used to specify what the system should do. [16]

F1 – User search

The search module can search among users.

F2 – Assignment search

The search module can search among assignments.

F3 – Submission search

The search module can search among submissions.

F4 – Exam search

The search module can search among exams.

F5 – Full-text search

The search module can full-text search.

F6 – Search among all document types by default

The search module searches among all document types, which were mentioned in F1, F2, F3 and F4, by default, but concrete document types can be specified using a tag.

F7 – Search among all attributes by default

The search module searches all the document's text attributes by default.

F8 – Incremental search

The search module can search incrementally, which means the unfinished query is used to find and present the relevant result.

F9 – *Basic* query

The search module must support searching by the terms a document is supposed to contain.

F10 – *Quoted* query

The search module must support queries where the order of terms in a query matters.

F11 – *Specific field* query

The search module must support searching by a specific field of document.

F12 – Boolean operators

The search module must support boolean operators (*and*, *or*, *not*) which can be used to join multiple queries, or negate the meaning of the query.

2.1.2 Non-functional requirements

Non-functional requirements are used to describe how the system performs a certain function. They are necessary to design appropriate system architecture. [16]

NF1 – Search performance

Since the search module must support incremental search (F8), search queries must be resolved in real-time (below 1 second).

NF2 – GraphQL API

The search functionality of the search module must be exposed using GraphQL API.

NF3 – Security

The search module must consider security. All of the functionality is available to teachers and administrators. Students can only search for their own submissions.

NF4 – LearnShell integrability

The search module must be compatible with LearnShell.

NF5 – Web UI

The search module's functionality must be accessible using an easy-to-use web interface integrated into LearnShell's site.

2.2 Full-text search

Full-text search provides the ability to search the whole content of natural-language documents. Given a query, which usually consists of the terms we want the document to contain, all the documents that satisfy the query are returned, usually sorted by relevance to the query. Searching by going through each word for each document would be highly inefficient, which led to the usage of indices.

Indexing is a process where the content of the document is preprocessed resulting in a helpful data structure (index) which is then used instead of the document to search rapidly. Preprocessing consists of the following three steps:

1. Parsing the document into tokens. It is useful to split the content of the document into different classes of tokens, such as numbers, words, and email addresses, so they could be processed differently.
2. Processing the tokens. Depending on the configuration, tokens are usually converted from upper-case to lower-case, the stopping words, which are words too common to be useful for searching, are removed and words are normalized by removing suffixes.
3. Storing the processed tokens. Finally, all of the processed tokens are stored in a data structure optimized for searching. [1]

	<u>term</u>	<u>freq</u>	<u>documents</u>
1: Winter is coming.	choice	1	3
2: Ours is the fury.	coming	1	1
3: The choice is yours.	fury	1	2
	is	3	1, 2, 3
	ours	1	2
	the	2	2, 3
	winter	1	1
	yours	1	3
	Dictionary		Postings

Figure 2.1: Indexing documents and resulting index (Source: [17])

In figure 2.1 an example of indexing three simple documents and the resulting data structure called inverted index can be seen. The index then could be used to find documents that contain the given token quickly since the tokens are sorted.

2.3 Data storage

The first thing to consider is whether the current LearnShell's data storage (PostgreSQL) is suitable, given the specified requirements. In this section, PostgreSQL's functionality along with two other alternatives is analyzed.

2.3.1 PostgreSQL

Relational databases are usually normalized and LearnShell is not an exception. Data in such a database are structured in a way following normal forms, achieving reduced data redundancy and improved data integrity. The cost of this is that whenever we want to query some related tables, they must be joined beforehand, which leads to some performance overhead. Such performance is not negligible at the moment when a large number of complex search queries must be handled in real-time.

Without further configuration, whenever a table is searched, the database has to scan the whole table to find the matching entries. This can be improved by creating an index on the given column which makes the search fast. The cost of this is that the insert, update and delete operations become slower as the index must be also changed. Since all the text fields must be searchable, indices would have to be created for each of them, resulting in another significant performance overhead. [1]

On the other hand, PostgreSQL provides support for efficient full-text searching functionality, which includes full-text indexing, linguistic support, and search result ranking. The disadvantage is that indices can be created above one table only, so fields from different tables cannot be indexed together and the first-mentioned problem occurs once again. [1]

2.3.2 Elasticsearch

Elasticsearch is the most popular open-source search engine built on top of Apache Lucene. [18] It provides features of a NoSQL database, such as horizontal scaling, schema-less data model, and an ability to handle a large amount of data. However, its main advantage is the rich searching functionality, including full-text search, powerful query DSL²², near real-time searching (below 1 second), and support of various data types. [19]

Regarding our use case, the data in such software are modeled based upon the queries. In contrast to relational databases, data redundancy is welcome, leading to avoidance of join operations and achieving high performance.

Elasticsearch is built to be horizontally scalable, which means deploying instances across multiple machines, achieving increased capacity, performance, and fault tolerance. [19]

²²Domain specific language

The functionality is exposed via a rich REST API and is completely JSON-based, which means all data which is supposed to be inserted into Elasticsearch must be converted to JSON beforehand, which is not a problem in LearnShell. Furthermore, external tools such as Logstash can be used to insert different data sources. [19]

Elasticsearch is usually used with two other products, Kibana and Logstash, named together as the ELK stack. Kibana is a web application used to explore, analyze and visualize the data in Elasticsearch. It can be especially helpful during development to debug. Logstash is a data aggregator which can be configured to automatically fetch data from various sources and send them to Elasticsearch. It is usually used to collect logs, but the plugins can be used to handle many different use cases. [20]

2.3.3 Apache Solr

Apache Solr is the second most popular open-source search engine. [18] The search engine is very similar to Elasticsearch in many ways, including being built on top of Apache Lucene. And that is the reason only the crucial differences will be mentioned.

Solr has always been more focused on enterprise-directed applications. It works well when searching a massive amount of static data is necessary and integrates easily with big data tools such as Hadoop and Spark. [19]

In contrast to Elasticsearch which supports JSON-based files only, Solr supports various data sources such as XML, CSV²³ files, Microsoft Word documents, and PDFs. [19]

Solr's documentation is much worse to become familiar with and is kind of lacking. Elasticsearch's one is much better organized, visualized, and offers great examples, therefore it's a clear winner in this case.

2.3.4 Summary

I mentioned some of the advantages and disadvantages of PostgreSQL, Elasticsearch, and Apache Solr. LearnShell does not need any of the enterprise-related functionality of Apache Solr and therefore Elasticsearch seems to be a better option. Even though PostgreSQL could handle the requirements of the search module, I decided to use both PostgreSQL and Elasticsearch. PostgreSQL remains the primary data storage and the only source of truth, whereas Elasticsearch becomes the secondary data storage with searchable data only. This solution ensures that the primary storage remains fast for all operations because the full-text search will be handled by Elasticsearch at the cost of data redundancy, additional service configuration, and the necessity of data synchronization.

²³Comma-separated values

2.4 Elasticsearch details

Since Elasticsearch has been chosen as the secondary data storage, it is necessary to explain some of its functionality a bit deeper.

As mentioned, Elasticsearch also works as a NoSQL database, which means the data are stored as JSON documents in the case of Elasticsearch. Since most people are rather familiar with relational database terminology, the comparison between them can be seen in Table 2.1.

Table 2.1: Comparison of Elasticsearch and RDBMS terminology

Elasticsearch	Relational database
Index	Database
Type	Table
Document	Record (Row)
Field	Attribute (Column)

It used to be possible to have several types for a single index, but because most of the configuration is on the index level, nowadays the best practice is to always have a single type for each index. Therefore, an index can be thought of as a collection of similar documents and each document is a collection of fields, which are the key-value pairs containing the data. By default, all fields are indexed and special data structured are created based upon the data type of the field. For example, for the text fields, inverted indices are created. [20]

2.4.1 Mapping

By definition: “*Mapping is the process of defining how a document, and the fields it contains, are stored and indexed.*”. [20] Elasticsearch is schema-less, which means mapping is being automatically done when new fields are encountered during inserting and data types are resolved from the data. However, it is often necessary to change the default behavior and therefore explicit mapping can be used to define the data.

Explicit mapping allows to precisely define the data type for each field of the document, customize analyzers for text fields and optimize fields for different kinds of queries. The same field can be indexed in different ways for different purposes. [20]

Listing 2 shows an example of a REST request to change the explicit mapping for index *assignments*. *Name* becomes a keyword field, *description* a text field and *inserted* a date field.

2.4.2 Analyzers

In Listing 2 two different field types were mentioned — keyword and text. The difference between them is that text field values are analyzed for full-

```
PUT /assignments
{
  "mappings": {
    "properties": {
      "name": {"type": "keyword"},
      "description": {"type": "text"},
      "inserted": {"type": "date"}
    }
  }
}
```

Listing 2: An example of explicit mapping

text search while keyword strings are left unchanged. An analyzer is used to preprocess the text and is composed of three building blocks:

Character filter

Firstly, the text is converted into a stream of characters. The stream is then processed by a character filter which can add, remove or change characters. For example, a character filter could be used to transform emoticons to their text equivalents (:) → happy). An analyzer can have zero or more character filters, which are applied in order.

Tokenizer

The processed stream of characters is split into individual tokens by a tokenizer. For example, whitespace tokenizer would split characters into tokens whenever it sees a whitespace. An analyzer must exactly have one tokenizer.

Token filters

Lastly, the stream of tokens is processed by a token filter which can add, remove or change individual tokens. The most common token filters are a lowercase token filter that converts all tokens to lowercase, a stop token filter that removes stop words, and a token filter to normalize words. An analyzer may have zero or more token filters. [20]

Elasticsearch offers several built-in analyzers which are often suitable enough for many different languages and types of text. However, the individual building blocks are also exposed, which can be used to build new custom analyzers specific to the use case. [20]

2.4.3 Query DSL

Query domain-specific language is provided by Elasticsearch to define queries in JSON and execute them using Elasticsearch's REST API. Some of the

queries relevant to the search module's requirements are explained:

Term query

The term query is the most basic type of query. It is used to search for a document that contains an exact term in the given field. Unlike full-text queries, an analyzer is not used.

Boolean query

The bool query is used to combine other queries using boolean operators.

Prefix query

The prefix query works similarly to the term query, but the query string is used as a prefix. Documents that contain the specified prefix in the given field are returned.

Match query

The match query is the standard query to perform a full-text search. An analyzer specified on the field is used to process the query string and then the documents are returned sorted by a score that corresponds to the relevance to the query. For example, the higher the occurrence of the terms in the document, the higher the score. Furthermore, in the case of several fields, some of them can have higher relevance than others which can be configured.

Multi-match query

The multi_match query is an extension of the match query which allows querying multiple fields.

Match boolean prefix query

The match_bool_prefix query analyzes its input but unlike the match query, constructs a bool query from the terms. Each term except the last one is used in a term query and the last one is used in a prefix query. It is a perfect query for implementing an incremental searching mentioned in the requirements.

Match phrase query

The match_phrase query works as the match query, but the given field must contain all terms and exactly after each other.

Match phrase prefix query

The match_phrase_prefix query works exactly as the match_phrase query, but the last term is used in a prefix query matching any words that begin with the term. [20]

2.5 Design

The section describes the key parts of the design of the search module. Whenever possible, diagrams are used to achieve a clearer way of description.

2.5.1 Data model diagram

Figure 2.2 describes the data model of the search module. For each searchable entity, an index will be created in Elasticsearch with the mentioned fields. It can be seen that some denormalization has to be done. Submissions newly contain a correction object to represent that the submission has been already corrected, a submitter username, an assignment name, and a generated assignment id. All of this to achieve maximum performance avoiding join operations at the cost of some memory.

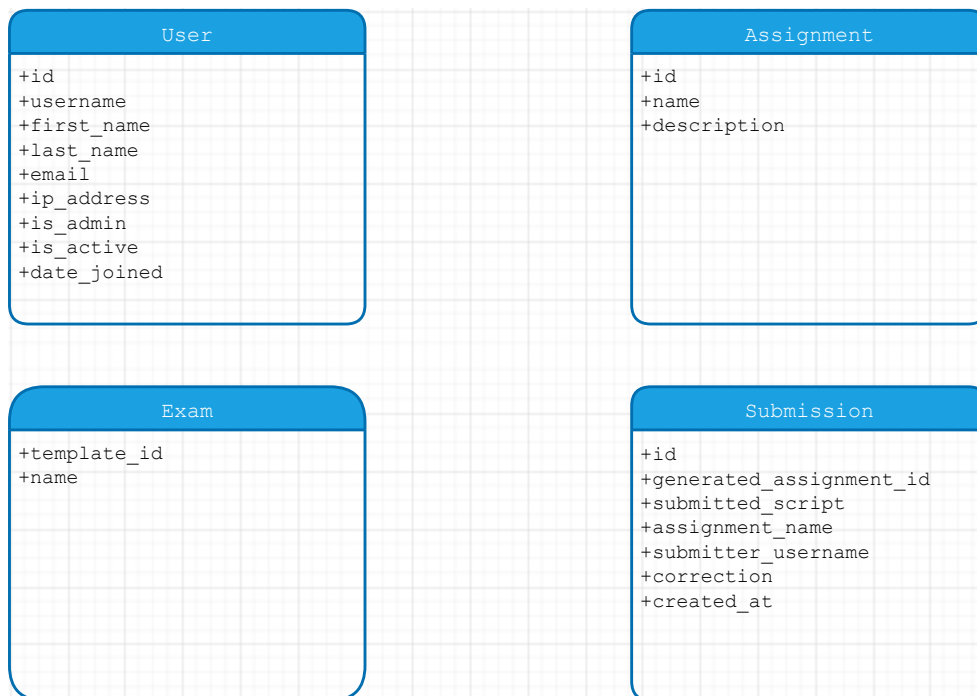


Figure 2.2: The search module data model (Created using [15])

2.5.2 GraphQL API

The GraphQL API closely corresponds to the data model. For each index, a GraphQL type is created with the same fields and with an argument for the query string. An example for an assignment using the GraphQL DSL is

2. ANALYSIS AND DESIGN

shown in Listing 3. `UserIndex` receives an argument representing the query string, which is then processed, executed and a list of the relevant documents is returned.

```
type AssignmentDocument {
  id: ID!
  name: String!
  description: String!
}

type Query {
  userIndex(query: String): [AssignmentDocument]!
}
```

Listing 3: GraphQL API definition example

2.5.3 The search module

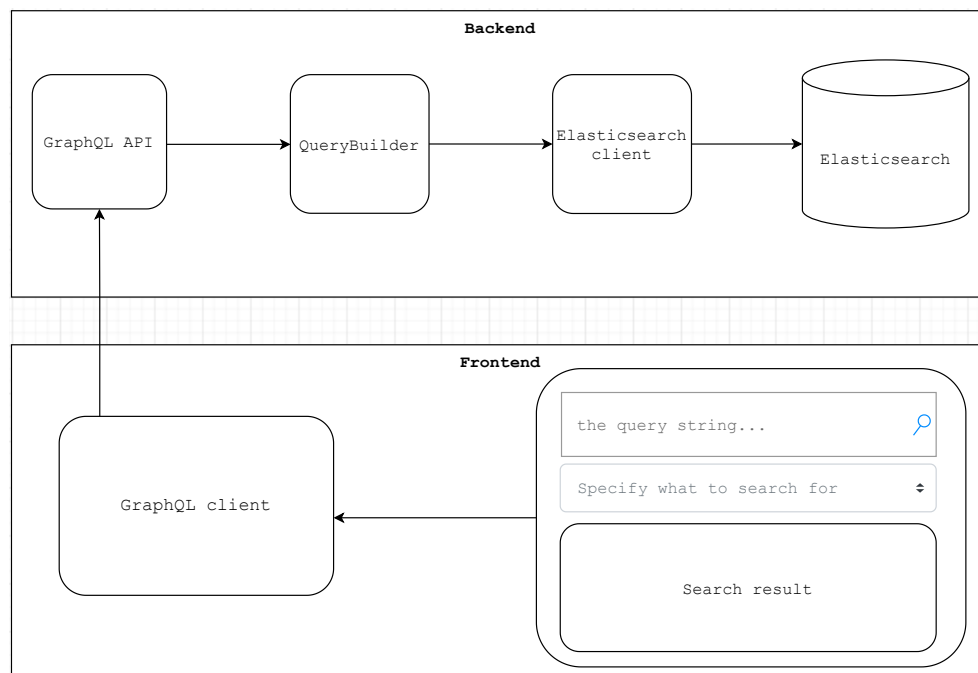


Figure 2.3: The search module architecture (Created using [15])

The diagram in Figure 2.3 shows all of the components of the search module and the communication between them. Each of the components has the

following meaning:

Search box component

The search box component is the main front-end component with which a user interacts. It is composed of three child components: a search box where a query string is written, a select box that can be used to specify a document type, and the search result component where the result is dynamically rendered. Whenever the query string is changed, it is immediately passed to the GraphQL client component.

GraphQL client

The GraphQL client receives the query string and the data from the select box builds an appropriate GraphQL query and executes the query. The result of the query is then returned to the search box component.

GraphQL API

The GraphQL API component is an implementation of the GraphQL server runtime and contains the index types mentioned in the previous section. It passes the query string to the QueryBuilder component.

QueryBuilder

The QueryBuilder component parses the query string and builds an appropriate Elasticsearch query using the Elasticsearch client's interface. The query is then executed and the result is returned.

Elasticsearch client

The Elasticsearch client component takes care of three things: synchronizes the data between the primary storage and Elasticsearch, defines the explicit mapping of indices, and exposes the search interface.

Realization

3.1 Back end

3.1.1 Elasticsearch, Kibana

First of all, it is necessary to configure the project's Docker compose file which can be located in the *ls* directory, to run Elasticsearch and Kibana services. Kibana was used during the development for debugging purposes such as viewing the data, removing indices, and using Kibana's command line to execute REST API commands.

Listing 4 shows the configuration that has been added to the Docker compose file for both newly added services. It can be seen that the Elasticsearch service exposes its REST API using port 9200, port 9300 is used for node communication which is not used in this case since only a single node is executed. Additionally, the Docker volume is configured to make Elasticsearch's data persistent across restarts. The Kibana web interface can be accessed at the localhost using port 5601.

3.1.2 Architecture

The following step is to implement the search module components in the back-end part of LearnShell. There are two possible ways to proceed:

1. The components may be implemented as part of a newly created microservice. This solution would achieve the maximum amount of decoupling and separating concerns at the cost of increased complexity. It would be necessary to implement additional API for the service and the data synchronization would be harder.
2. The components may be implemented as part of LearnShell's Django project. This solution would make the data synchronization easy since direct access to Django's ORM could be made use of.

3. REALIZATION

```
ls-elasticsearch:
  networks:
    - ls-bridge
  image: docker.elastic.co/elasticsearch/elasticsearch:7.11.1
  volumes:
    - 'ls-elasticsearch-data:/usr/share/elasticsearch/data'
  environment:
    - node.name=es01
    - cluster.name=es-docker-cluster
    - discovery.zen.minimum_master_nodes=1
    - discovery.type=single-node
    - network.host=0.0.0.0
    - bootstrap.memory_lock=true
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  ulimits:
    memlock:
      soft: -1
      hard: -1
  ports:
    - 9200:9200
    - 9300:9300
ls-kibana:
  networks:
    - ls-bridge
  image: docker.elastic.co/kibana/kibana:7.11.1
  environment:
    ELASTICSEARCH_URL: http://ls-elasticsearch:9200
    ELASTICSEARCH_HOSTS: '["http://ls-elasticsearch:9200"]'
  ports:
    - 5601:5601
  depends_on:
    - ls-elasticsearch
```

Listing 4: Elasticsearch and Kibana docker compose configuration

The second option has been chosen since the advantage of having easier data synchronization is crucial. Furthermore, Django provides a way to structure the project into separate, reusable modules called applications. This way, the codebase would not be polluted with the search module code. Therefore, the *smart_search* directory was created in the *apps* directory where all back-end components will be located.

Another consequence of the chosen solution is that all code will be written in Python. The code will be written with the emphasis put on Python idioms to make maximum use of the language. Furthermore, even though the language is dynamically typed, type hints will be used to make the code more clear and avoid some of the bugs in advance.

3.1.3 Elasticsearch client

The Elasticsearch client component is built using a Python library — Django Elasticsearch DSL²⁴. It is a thin wrapper around `elasticsearch-dsl-py`²⁵ library which provides all features of Elasticsearch DSL but in a pythonic object-oriented API avoiding the necessity to directly use Elasticsearch’s REST API. Additionally, it provides support for Django ORM models and allows data synchronization via hooks to Django model methods.

Listing 5 shows an example of an index declaration for submissions using the library. A Python class is used to specify all necessary information in a declarative way, which is then extracted using reflection. Each class attribute with the given configuration is then used to create the explicit mapping of the given index. Methods that start with the string “prepare” are called before the corresponding Django model data are inserted and can be used to preprocess the data. In the example, they are used to decode the base64 strings and for denormalization purposes.

The `full_text_entrypoint` field is a custom one to which all other fields are copied. This way, all fields can be queried in a single query instead of querying each field separately. The `SearchAsYouType` data type is specialized for incremental searching. The details can be found in [20].

Whenever a Django ORM object is created, updated, or deleted, the corresponding Elasticsearch index is automatically changed in a non-blocking way, which is necessary because indexing can take some time. The class also provides methods to cover all Elasticsearch queries. And in a similar way, all searchable entities are covered.

3.1.4 Query builder

As mentioned, the query builder component’s task is to parse the query string and produce the corresponding Elasticsearch query. The component is built upon a Python library — `PyParsing`²⁶. The library is used to define a parsing expression grammar (PEG) and based upon the grammar, provides parsing functionality. PEGs are an alternative to context-free grammars having the same expressive power but are easier to use for describing machine-oriented languages. [21]

Listing 6 shows a class containing the grammar definition which is declared using class variables to avoid building a new parser for each call. Furthermore, a dictionary is used as a cache to avoid reparsing the same query strings.

²⁴<https://github.com/django-es/django-elasticsearch-dsl>

²⁵<https://github.com/elastic/elasticsearch-dsl-py>

²⁶<https://pyparsing-docs.readthedocs.io/en/latest/>

3. REALIZATION

```
@registry.register_document
class SubmissionDocument(Document):
    """
    Contains Submission Elasticsearch index definition and mapping to corresponding Django model
    class.
    ↪ Fields are 1:1 mapped to corresponding Django model objects (can be changed using attr='
    ↪ parameter).
    """
    # Fields with with explicit data type mapping
    submitted_script = fields.TextField(attr='submission_data', copy_to=SEARCH_AS_YOU_TYPE_FIELD,
    ↪ analyzer=STANDARD_ANALYZER_WITH_ASCII_FOLDING)
    assignment_name = fields.TextField(copy_to=SEARCH_AS_YOU_TYPE_FIELD,
    ↪ analyzer=STANDARD_ANALYZER_WITH_ASCII_FOLDING)
    submitter_username = fields.TextField(copy_to=SEARCH_AS_YOU_TYPE_FIELD,
    ↪ analyzer=STANDARD_ANALYZER_WITH_ASCII_FOLDING)
    correction = fields.ObjectField(properties={
        'id': fields.IntegerField(),
        'score': fields.IntegerField()
    })

    def prepare_assignment_name(self, submission_instance: Submission):
        gen_assignment = submission_instance.generated_assignment
        assignment_name = gen_assignment.assignment.name
        return assignment_name

    def prepare_submitter_username(self, submission_instance: Submission):
        gen_assignment = submission_instance.generated_assignment
        student_username = gen_assignment.student.username
        return student_username

    def prepare_submitted_script(self, submission_instance: Submission):
        data_dict = submission_instance.submission_data
        script_base64 = data_dict['script']
        return decode_base64(script_base64)

    generated_assignment_id = fields.LongField()
    full_text_entrypoint = SearchAsYouType(analyzer=STANDARD_ANALYZER_WITH_ASCII_FOLDING)

class Index:
    """
    Contains Elasticsearch index related configuration.
    """
    # Name of the Elasticsearch index
    name = 'submissions'
    # See Elasticsearch Indices API reference for available settings
    settings = {'number_of_shards': DEFAULT_NUMBER_OF_SHARDS,
               'number_of_replicas': DEFAULT_NUMBER_OF_REPLICAS}

class Django:
    """
    Contains Elasticsearch index - Django model mapping configuration.
    """
    model = Submission # The model associated with this Document

    # The fields of the model you want to be indexed in Elasticsearch with implicit data type
    ↪ mapping
    fields = [
        'id',
        'created_at'
    ]
```

Listing 5: Submission index declaration

```

class QueryBuilder:
    """
    QueryBuilder class contains grammar definition for pyparsing module
    and single method which parses input strings and generates Elasticsearch
    ↪ query.
    """
    # Cache for built queries
    builder_cache: Dict[str, Search] = {}
    # Grammar definition using pyparsing module
    not_operator = pp.Suppress(pp.CaselessKeyword('not') | pp.Literal('~'))
    and_operator = pp.Suppress(pp.CaselessKeyword('and') | pp.Literal('&'))
    or_operator = pp.Suppress(pp.CaselessKeyword('or') | pp.Literal('|'))
    quoted_str = pp.QuotedString(quoteChar='"', unquoteResults=True)
    specific_field_query = pp.Combine(pp.Word(pp.alphanums +
    ↪ '._')('field_key') + pp.Literal(':') + quoted_str('field_value'))
    quoted_query = quoted_str
    words_query = pp.Group(pp.ZeroOrMore(~(and_operator | or_operator |
    ↪ not_operator) + pp.Word(pp.printables)))
    query = quoted_query | specific_field_query | words_query
    grammar = pp.infixNotation(query, [(not_operator, 1, pp.opAssoc.RIGHT,
    ↪ handle_not_operator),
                                     (and_operator, 2, pp.opAssoc.LEFT,
    ↪ handle_and_operator),
                                     (or_operator, 2, pp.opAssoc.LEFT,
    ↪ handle_or_operator)])
    specific_field_query.setParseAction(handle_specific_field_query)
    quoted_query.setParseAction(handle_quoted_query)
    words_query.setParseAction(handle_words_query)

    @staticmethod
    def build_query(search_query: str) -> Search:
        """
        Parses input string and generates corresponding Elasticsearch query.
        :param search_query: String representing search query
        :return: Elasticsearch Search object generated from input string
        """
        # Look at cache
        if search_query in QueryBuilder.builder_cache:
            return QueryBuilder.builder_cache[search_query]

        parser_result = QueryBuilder.grammar.parseString(search_query)
        [elastic_query] = parser_result
        QueryBuilder.builder_cache[search_query] = elastic_query
        return elastic_query

```

Listing 6: QueryBuilder class

3.1.5 GraphQL API

The GraphQL API of LearnShell is generated by an external Python library that is tightly coupled to the Django ORM models and this caused a problem of not having enough flexibility to declare the new search API because custom objects with custom arguments had to be made. To avoid modifying the internals of the library, a new GraphQL endpoint “/graphql_search”, next to the original “/graphql”, was built just to provide the searching functionality.

A Python library Graphene²⁷ which provides all of GraphQL’s standard functionality and implements the runtime, was used to build the new API. An example of the declaration of index types can be seen in Listing 7. Each type takes a query string as an argument and returns a list of the corresponding documents. Resolvers are used to handle the task of getting the data. It can be seen that the dispatch function takes a user object to handle the security, type of index to be searched, and the query string. And it returns a list of documents matching the query sorted by relevance.

```
class QueryType(graphene.ObjectType):
    """
    Contains Query type definition for GraphQL API.
    Entrypoint for all immutable GraphQL queries.
    """

    user_index = graphene.List(UserDocumentType, **QUERY_PARAMS,
        ↪ required=True)
    assignment_index = graphene.List(AssignmentDocumentType, **QUERY_PARAMS,
        ↪ required=True)
    submission_index = graphene.List(SubmissionDocumentType, **QUERY_PARAMS,
        ↪ required=True)
    exam_index = graphene.List(ExamDocumentType, **QUERY_PARAMS,
        ↪ required=True)

    # Resolvers handle GraphQL queries for above fields.
    def resolve_user_index(root, info, query):
        return dispatch_search_request(info.context.user, UserDocument, query)

    def resolve_assignment_index(root, info, query):
        return dispatch_search_request(info.context.user, AssignmentDocument,
            ↪ query)

    def resolve_submission_index(root, info, query):
        return dispatch_search_request(info.context.user, SubmissionDocument,
            ↪ query)

    def resolve_exam_index(root, info, query):
        return dispatch_search_request(info.context.user, ExamDocument, query)
```

Listing 7: GraphQL API declaration example

²⁷<https://graphene-python.org/>

3.2 Front end

The front end's components will be part of LearnShell's Next.js project and will be written in TypeScript. To follow the project's structure, React components will be put into the components directory, whereas pure TypeScript code will be put into the modules directory.

3.2.1 GraphQL client

First of all, it is necessary to be able to connect to the newly built GraphQL API. A lightweight client `graphql-request`²⁸ already used by LearnShell is good enough for this task.

In Listing 8 can be seen two main functions of the component. `BuildGraphQLQuery` takes care of generating a GraphQL query based upon the parameters to avoid fetching unnecessary entities. `HandleQuery` is an entry point to the component which receives a query string and takes care of all the steps to get to the actual search result of the query. The function is asynchronous which means a promise object is returned instead of waiting for the result and blocking the code flow resulting in a page freeze. A callback function can be passed to the promise which is called the moment the result is available.

```

async function handleQuery(searchQuery: string, filterList:
↳ Array<DocumentSelectOption>, allSelectOptions: Array<DocumentSelectOption>):
↳ Promise<SearchResult> {
  const graphqlQuery = buildGraphQLQuery(filterList, allSelectOptions);
  const searchQueryResult = await searchFetcher(graphqlQuery, {'query':
↳ searchQuery});
  return searchQueryResult;
}

function buildGraphQLQuery(filterList: Array<DocumentSelectOption>, allSelectOptions:
↳ Array<DocumentSelectOption>): string {
  if (filterList.length === 0) {
    filterList = allSelectOptions;
  }
  const innerQueries = filterList.map((docType) =>
↳ documentTypeQueryMapping[docType.value]).join('');
  const query = gql`query searchByGivenQuery($query: String!) {
↳ ${innerQueries}
↳ `
  return query;
}

```

Listing 8: GraphQL client example

²⁸<https://github.com/prisma-labs/graphql-request>

3.2.2 React components

React components are needed for building the UI of the search module. They will handle the user interaction and delegate the search requests to the GraphQL client component described in the previous section. LearnShell makes use of Atlassian's²⁹ official UI library when building React components which also provides components useful to the search module.

QuickSearch is an Atlassian component suitable to the requirements. It provides a search box with a place to render the search result. The best feature is that it takes care of the visual design and the component matches the rest of the page. The component was used to build an actual search module component that takes care of handling the search events, getting the search result, and rendering the search result in a useful way.

LearnShell's page has a side panel that can be used to contain the access point to the search component because it can be accessed at any time without redirection. The panel is represented by a drawer component which was therefore modified to contain the newly implemented search component. The components in action can be seen in Figure D.1.

²⁹<https://atlassian.com/>

Testing

Testing is a fundamental part of the development cycle of any software. It ensures that the requirements are truly fulfilled and minimizes the chance of the occurrence of bugs. The search module was tested during the development with one of the testing methods — unit tests.

4.1 Unit testing

Unit testing is about testing individual units of code such as functions, methods, or classes depending on the language and the programming paradigm. Those units are tested separated from the rest of the application meaning any external dependencies must be replaced with fake objects called mocks. This ensures that only the unit is being tested.

One of the advantages of this method is that unit tests can be written once and then run automatically by a testing framework at any time. Whenever the codebase changes, unit tests ensure that the change did not break any of the units of code.

On the other hand, it may be hard to design unit tests for complex code with lots of dependencies and unit tests that are actually useful. Furthermore, unit tests do not guarantee that the codebase is bug-free since it is not possible to cover all the possible execution paths of the code. And from the essence of the matter, they will not catch bugs that come from the integration of several units.

4.2 Python unit tests

For back-end components which were were written in Python, the language provides native support for unit tests — unittest³⁰. The testing framework

³⁰<https://docs.python.org/3/library/unittest.html>

4. TESTING

was used to cover the most important parts of the components. An example of such a unit test can be seen in Listing 9 where the authorization logic is being tested.

```
class TestHandleAuthorization(TestCase):
    def test_user_is_authorized(self):
        mocked_user = Mock(spec=User)
        mocked_user.is_admin = False
        res = handle_authorization(SubmissionDocument, mocked_user)
        self.assertIsInstance(res, Search)

    def test_user_is_not_authorized(self):
        non_authorized_documents = (UserDocument, ExamDocument,
        ↪ AssignmentDocument)
        mocked_user = Mock(spec=User)
        mocked_user.is_admin = False

        for doc in non_authorized_documents:
            with self.assertRaises(PermissionDenied):
                handle_authorization(doc, mocked_user)

    def test_admin_is_always_authorized(self):
        mocked_admin = Mock(spec=User)
        mocked_admin.is_admin = True
        res = handle_authorization(SubmissionDocument, mocked_admin)
        self.assertIsInstance(res, Search)

    def test_not_supported_document_type(self):
        mocked_document_type = Mock(spec=Document)
        mocked_user = Mock(spec=User)
        mocked_user.is_admin = False
        with self.assertRaises(NotImplementedError):
            handle_authorization(mocked_document_type, mocked_user)
```

Listing 9: Python unit test example

4.3 TypeScript unit tests

Since JavaScript or TypeScript do not come with native support for unit testing and neither LearnShell did use any, a new testing framework had to be chosen and integrated. Jest³¹ is the most popular one and also comes with an extension ts-jest³² for supporting TypeScript. [22] An example of a unit test using the Jest's API can be seen in the example 10 where an asynchronous function that takes care of executing GraphQL queries, is being tested. The API is quite similar to Python unit tests but is written in a functional way.

³¹<https://jestjs.io/>

³²<https://github.com/kulshekhar/ts-jest>

```

jest.mock('graphql-request')

describe('Test GraphQL api fetcher function', () => {
  test('Fetcher calls GraphQL client with correct parameters.', async () =>
    ↪ {
      const mockedGraphQLClientClass = (graphql.GraphQLClient as jest.Mock);
      const query = 'graphqlquery';
      const queryVariables = {};
      await searchFetcher(query, queryVariables);
      expect(mockedGraphQLClientClass.mock.instances[0]
        .request.mock.calls[0][0]).toBe(query);
      expect(mockedGraphQLClientClass.mock.instances[0]
        .request.mock.calls[0][1]).toBe(queryVariables);
    })
})

```

Listing 10: TypeScript unit test example

4.4 React component unit tests

To be able to write unit tests for React components using Jest, another tool is required. Enzyme³³ allows to shallowly render the components and simulate user interactions on them. The unit test coverage of the main component can be seen in Listing 11.

```

// Configure Enzyme to use React 16 adapter
Enzyme.configure({adapter: new Adapter()});

// React component shallow testing using Enzyme
describe('SearchDrawer React component tests', () => {
  test('Admin should have option to select document type', () => {
    const searchDrawer = shallow(<SearchDrawer admin={true}/>);
    expect(searchDrawer.find('CheckboxSelect').exists()).toBeTruthy();
  })
  test('Students dont have option to select document type', () => {
    const searchDrawer = shallow(<SearchDrawer admin={false}/>);
    expect(searchDrawer.find('CheckboxSelect').exists()).toBeFalsy();
  })
  test('Searching actually changes value of search box because it is controlled
    ↪ component', () => {
    const searchDrawer = shallow(<SearchDrawer admin={false}/>);
    const searchQuery = "testQuery";
    // Simulate searching event
    searchDrawer.simulate('SearchInput', {
      target: {value: searchQuery}
    });
    expect(searchDrawer.prop('value')).toBe(searchQuery);
  })
})

```

Listing 11: React component unit test example

³³<https://enzymejs.github.io/enzyme/>

Conclusion

The goal of the thesis was to come up with a smart search module for LearnShell, which will improve the user search experience. This goal has been successfully achieved following the common software engineering development cycle: analyze, design, implement and test.

The whole LearnShells's infrastructure was described, including its technologies and some of their advantages and disadvantages, the data model and some of the important entities, the microservice architecture, and — finally — its original searching functionality.

The functional and non-functional requirements of the search module were specified. The appropriate database storage was analyzed which led to the selection of Elasticsearch and finally the changes to the backend and frontend of LearnShell were designed.

The realization part of the search module was described. The maximum effort was held to use the correct tools and patterns which positively influenced the whole development process and made many things much easier.

In the last chapter, the unit testing method was described and shown in the examples of unit tests that were used during the development.

Finally, some of the ideas by which the module could be improved, are suggested:

- Front-end search box could have some highlighting features for characters with special meaning. This would require the parser to send the parsed query together with the search result, or the frontend would have its own parser.
- At this time, the search module is fault-tolerant, but without any feedback. If there is an error, it just returns an empty result set. This could be improved by informing the user why the query failed. For example, when a non-existing field is accessed.

CONCLUSION

- There is no way to change the priority of *and* and *or* operators, also they cannot be used to form more complex nested queries. The parser could be improved by nested bool queries using parentheses.

Bibliography

- [1] “About PostgreSQL”. In: *PostgreSQL.org [online]* (1996-2021). [cit. 2021-04-22]. URL: <https://www.postgresql.org/about/>.
- [2] In: *Djangoproject.com [online]* (2005-2021). [cit. 2021-04-21]. URL: <https://www.djangoproject.com/start/overview/>.
- [3] “Object-relational Mappers (ORMs)”. In: *Full Stack Python [online]* (2012-2021). [cit. 2021-04-22]. URL: <https://www.fullstackpython.com/object-relational-mappers-orms.html>.
- [4] “What is REST”. In: *Restfulapi.net [online]* (2020). [cit. 2021-04-22]. URL: <https://restfulapi.net/>.
- [5] In: *GraphQL.org [online]* (2021). [cit. 2021-04-23]. URL: <https://graphql.org/>.
- [6] “GraphQL Advantages and Disadvantages”. In: *Javatpoint.com [online]* (2011-2018). [cit. 2021-04-23]. URL: <https://www.javatpoint.com/graphql-advantages-and-disadvantages>.
- [7] “Single-Page Apps vs Multiple-Page Web Apps: What to Choose for Web Development”. In: *Yalantis.com [online]* (2021). [cit. 2021-04-23]. URL: <https://yalantis.com/blog/single-page-apps-vs-multiple-page-apps/>.
- [8] “React documentation”. In: *Reactjs.org [online]* (2021). [cit. 2021-04-23]. URL: <https://reactjs.org/docs>.
- [9] “Next.js documentation”. In: *Nextjs.org [online]* (2021). [cit. 2021-04-25]. URL: <https://nextjs.org/docs>.
- [10] “Typescript documentation”. In: *Typescriptlang.org [online]* (2012-2021). [cit. 2021-04-24]. URL: <https://www.typescriptlang.org/docs/>.
- [11] “Docker documentation”. In: *Docker.com [online]* (2013-2021). [cit. 2021-04-25]. URL: <https://docs.docker.com/>.

- [12] “The Docker Handbook – 2021 Edition”. In: *Freecodecamp.org [online]* (2021). [cit. 2021-04-25]. URL: <https://www.freecodecamp.org/news/the-docker-handbook/>.
- [13] Kernel Korner. “Unionfs: Bringing Filesystems Together”. In: *Linuxjournal.com [online]* (2004). [cit. 2021-04-25]. URL: <https://www.linuxjournal.com/article/7714>.
- [14] Chris Richardson. “Pattern: Microservice Architecture”. In: *Microservices.io [online]* (2018). [cit. 2021-04-21]. URL: <https://microservices.io/patterns/microservices.html>.
- [15] In: *Diagrams.net [online]* (2021). [cit. 2021-04-21]. URL: <https://app.diagrams.net/>.
- [16] Ulf Eriksson. “Why is the difference between functional and Non-functional requirements important?” In: *Reqtest.com [online]* (2012). [cit. 2021-04-28]. URL: <https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements>.
- [17] Alex Brasetvik. “Elasticsearch from the Bottom Up, Part 1”. In: *Elastic.co [online]* (2013). [cit. 2021-04-29]. URL: <https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up>.
- [18] “DB-Engines Ranking of Search Engines”. In: *Db-engines.com [online]* (2021). [cit. 2021-04-30]. URL: <https://db-engines.com/en/ranking/search+engine>.
- [19] “Solr vs. Elasticsearch: Who’s The Leading Open Source Search Engine?” In: *Logz.io [online]* (2020). [cit. 2021-04-30]. URL: <https://logz.io/blog/solr-vs-elasticsearch>.
- [20] “Elasticsearch documentation”. In: *Elastic.co [online]* (2021). [cit. 2021-04-30]. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/>.
- [21] Bryan Ford. “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation”. In: *pdos.csail.mit.edu [online]* (2004). [cit. 2021-05-01]. URL: <https://pdos.csail.mit.edu/papers/parsing:popl04.pdf>.
- [22] Testim. “What Is the Best Unit Testing Framework for JavaScript?” In: *Testim.io [online]* (2021). [cit. 2021-05-05]. URL: <https://www.testim.io/blog/best-unit-testing-framework-for-javascript/>.

Acronyms

- API** Application programming interface
- DRY** Don't repeat yourself
- DSL** Domain specific language
- GUI** Graphical user interface
- HTML** HyperText Markup Language
- HTTP** Hypertext transfer protocol
- JSON** JavaScript object notation
- MVC** Model-view-controller
- ORM** Object-relational mapping
- RDBMS** Relational database management system
- REST** Representational state transfer
- SPA** Single-page application
- UI** User interface

Contents of enclosed CD

	readme.txt.....	the file with contents description
	src.....	the directory of source codes
	doc.....	the directory of user documentation sources
	impl.....	LearnShell implementation sources
	thesis.....	the directory of L ^A T _E X source codes of the thesis
	text.....	the thesis text directory
	thesis.pdf.....	the thesis text in PDF format
	doc.pdf.....	the user documentation in PDF format

APPENDIX **C**

Figures

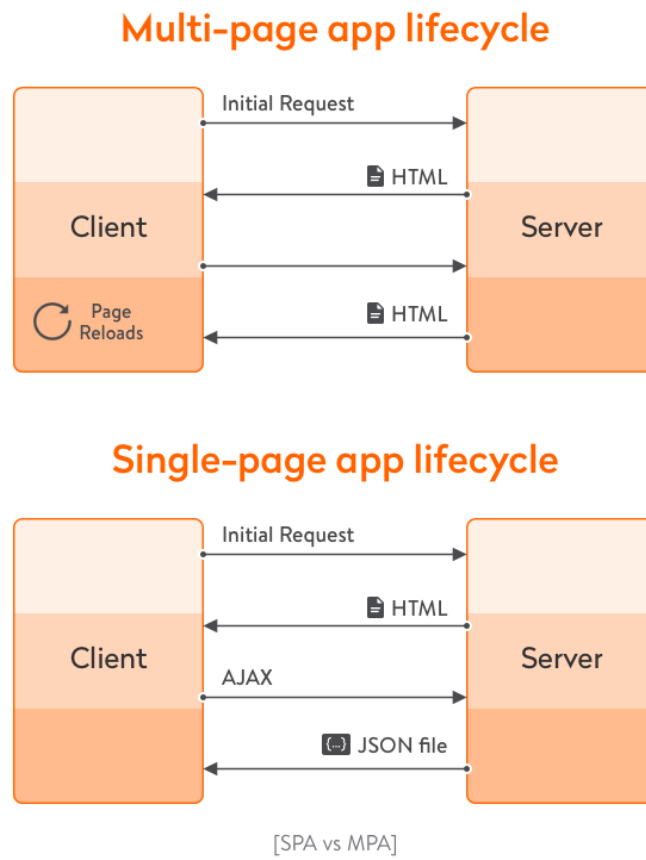


Figure C.1: Difference between SPA and traditional web page [7]

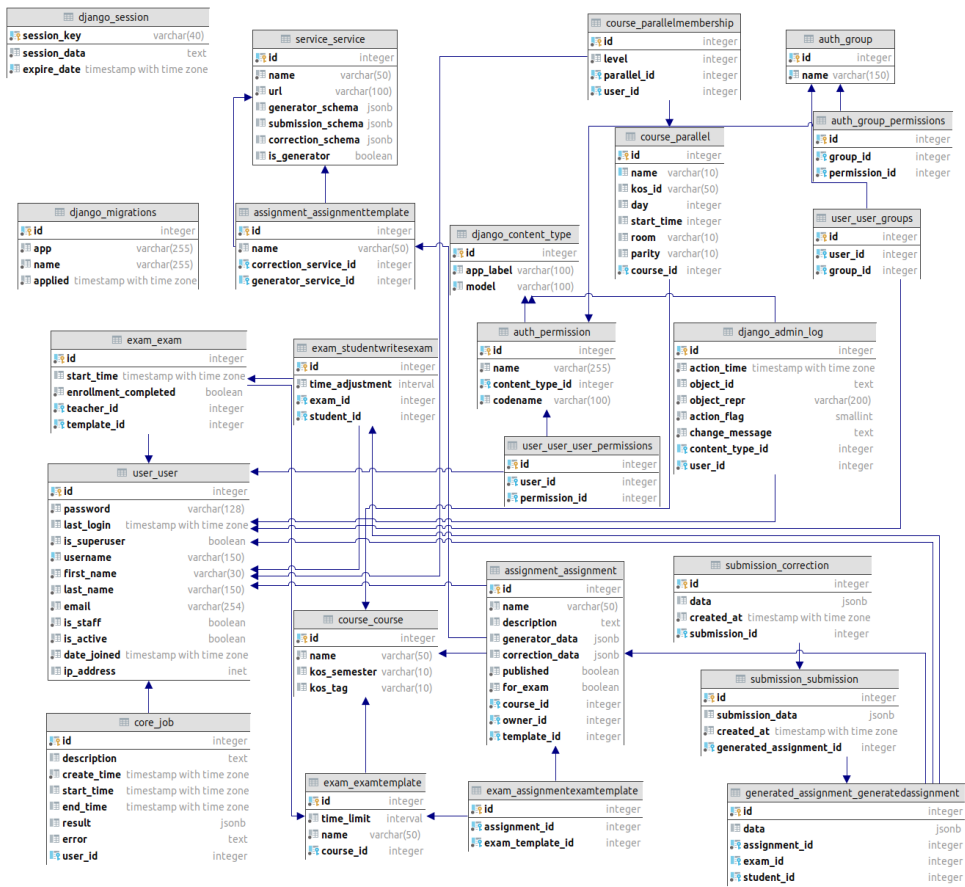


Figure C.2: LearnShell database diagram (Created in PyCharm)









Result examples

D. RESULT EXAMPLES

← du ceny akc|

Searching for...

ASSIGNMENTS

-  2020-11 DU Ceny akcií Napište skript, který přijímá dva parametry. **První p... assignment
-  en-2020-09 DU Change of the file format to csv The www.barchart.com ... assignment
-  2020-11 DU Stock prices Write a script that accepts two parameters. **The ... assignment
-  2020-09 DU Změna formátu souboru na csv Server www.barchart.com o... assignment
-  en-2020-09 DU Word with the highest frequency The directory name is ... assignment
-  en-2020 DU Censorship Write a script that accepts one parameter, which co... assignment
-  2020-09 DU Nejčtetnější sedmiznakové slovo V proměnné **DIR** je ulož... assignment
-  2020-10 DU Cenzura Napište skript, který přijímá jeden parametr, ve kterém ... assignment

SUBMISSIONS











-  2020-11 DU Ceny akcií (GJVMZNMfvspRdw) /a submission
-  2020-11 DU Ceny akcií (mLpBQVv3l8HjCyp) submission
-  2020-11 DU Ceny akcií (joZTQp9ALU06ntn) # submission
-  2020-11 DU Ceny akcií (EqF8f2oU8ykMV0g) . submission
-  2020-11 DU Ceny akcií (w8VXU03PZYmgJq5) 1 submission
-  2020-11 DU Ceny akcií (mOkyatqxHB6rSV4) ls submission
-  2020-11 DU Ceny akcií (w8VXU03PZYmgJq5) 111 submission
-  2020-11 DU Ceny akcií (0mhTxWmR2vr0mh2) xd submission
-  2020-11 DU Ceny akcií (5i1jLYvaVgj4ws3) vchg submission
-  2020-11 DU Ceny akcií (PPjT5j8d5E33dMb) awk submission

Figure D.1: A basic query example

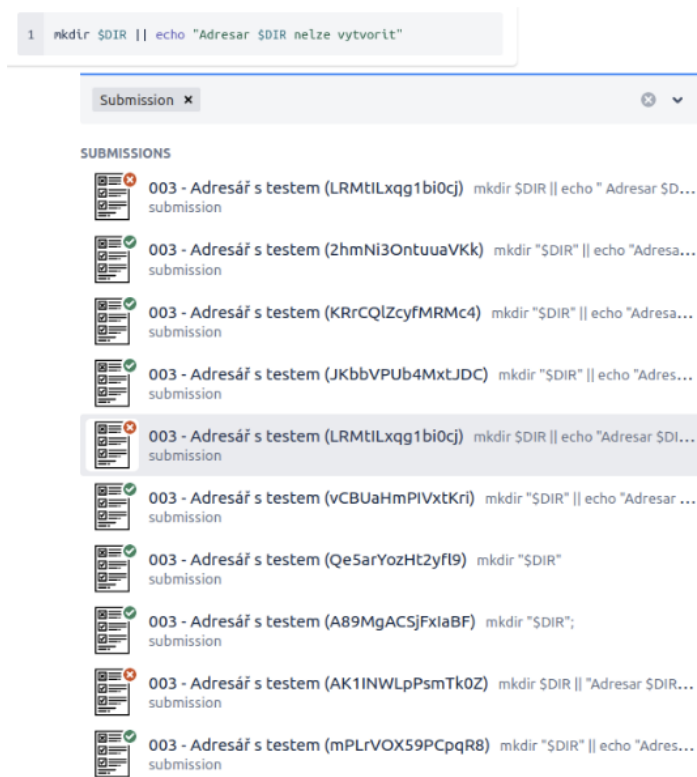


Figure D.2: A tooltip example for submission script that does not fit (same works for assignment descriptions)

D. RESULT EXAMPLES

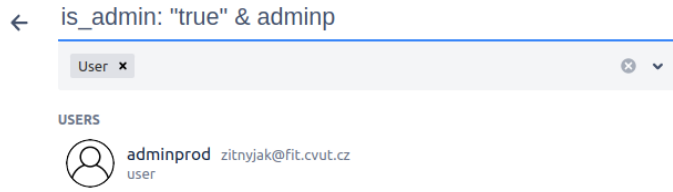


Figure D.3: An example of specific field query and *and* operator

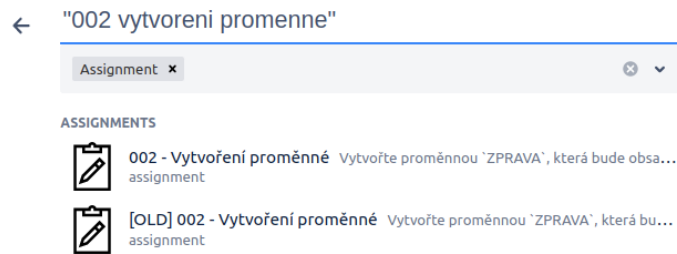


Figure D.4: An example of quoted query

D. RESULT EXAMPLES

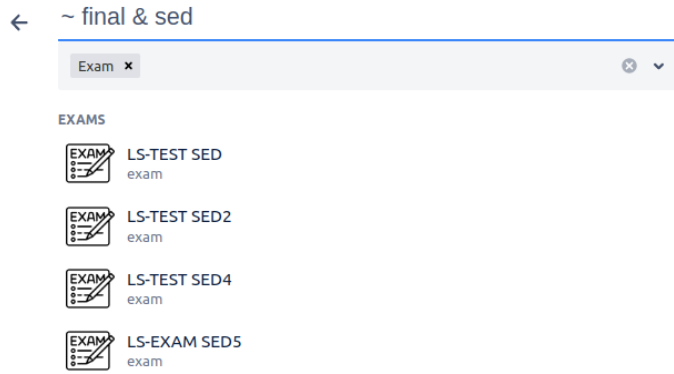


Figure D.5: An example of *not* operator and *and* operator

User documentation

E.1 SmartSearch - User documentation

SmartSearch is Learnshell's module to improve user experience whenever finding some of the documents is necessary. It provides an easy to use search box which returns the most suitable result set just as you type the query in real time. All of its features with examples are explained in the following sections.

E.1.1 Authorization

The module has been developed with primarily teachers on mind, which can search all of the document types. Students can only search their own submissions.

E.1.2 Searchable documents

SmartSearch allows you to search users, submissions, assignments and exams. Each document is searched using all its text and keyword fields. Based upon the type, explained in the following section, the query is parsed, analyzed and executed. Documents are returned sorted based upon how much the document matches the query.

- ***Users***
 - username (Text field)
 - first_name (Text field)
 - last_name (Text field)
 - email (Keyword field)
 - is_admin (Boolean field)
 - is_active (Boolean field)
 - date_joined (Date field)
- ***Assignments***
 - name (Text field)
 - description (Text field)
- ***Submissions***
 - assignment_name (Text field)
 - submitted_script (Text field)
 - submitter_username (Text field)
 - correction (Object field)
 - * score (Integer field)
 - created_at (Date field)
- ***Exams***
 - name (Text field)

E.1.3 Query types

The following query types are supported:

- **Basic query** - Is used as default query. It's a string of arbitrary characters including punctuation and white spaces. Except those keywords and characters which modify the query as explained later: **and**, **&**, **or**, **|**, **not**, **~**. During the execution the query is split into terms and each term except the last one must be included in any order in the document to be matched. The last term is matched using its prefix.

Examples:

- `create two subdirectories.` - Matches documents which contains terms: create, two and subdirectories.
- `forget the hidden fi` - Matches documents which contains terms: forget, hidden and any term with prefix *fi*. "The" is removed as it is an english stop word.

- **Quoted query** - Arbitrary string enclosed in double quotes (""). It's used when the order of terms matter, unlike for basic queries or whenever you want to have a term with special meaning (and, or, ...) as part of the query. Backslash (\) can be used to escape embedded double quotes.

Examples:

- `"files and directories in \"${PATH}\"` - Looks for documents with following phrase: *files and directories in "\${PATH}"*.

- **Specific field query** - It's used when we want the field to have exact value. The query must have the following format: `<field_name>:<zero_or_more_whitespaces> "<value>"`. Any field mentioned in the Searchable documents section which it makes sense for can be used. Sub-fields can be accessed using dot notation as shown in following examples.

Examples:

- `is_admin:"true"` - Returns all users with admin privileges.
- `correction.score: "1"` - Returns submissions which have been corrected and received 1 point.
- `username: "user1337"` - Returns user user1337.

E.1.4 Bool operators

Bool operators can be used to join or negate the meaning of several queries. Nesting operators using parantheses is not supported, but might be added in the future if found necessary.

Precedence: *not* > *and* > *or*

Left association - and, or

Right association - not

- **And operator** - All queries must be matched by the document to be returned. Format: `<query1> <and, &> <query2> [<and|&> <query3>, ...]`

Examples:

- `file and directory & path` - Three basic queries are created for each term (file, directory, path) and all must be matched by the document. The difference is that each term is used as prefix. Whereas in corresponding basic query (`file directory path`) only the last term would be used as prefix.
- `"Assignment 2" and "variable modification"` - Creates two quoted queries and both must be matched by the document.

- **Or operator** - At least one of the queries must be matched by the document to be returned. Format: `<query1> <or, |> <query2> [<or, |> <query3>, ...]`

Examples:

- `Assignment 2 | Assignment 3 or assignment 4` - Creates three basic queries and at least one of them must be matched by the document.

- **Not operator** - The query must not be matched by the document to be returned. Format: `<not, ~> <query>`

Examples:

- `~ correction.score: "0"` Negates the meaning of the quoted query. Returns all document which have gained at least 1 point.
- `not "Final Exam 3" and exam` - Returns all exams except the final exam 3.
- `not count some numbers` - Not can also be used with basic queries if it is the first term in the query. Document must not be matched by terms: *count*, *some* and prefix *numbers*.