

Mika Hämäläinen

INTEGRATING OPENSEARCH WITH A GIT-BASED DOCUMENTATION SYSTEM

Master's thesis
Faculty of Engineering and Natural Sciences
Examiner: University instructor Mikko Salmenperä
Examiner: Professor Matti Vilkkio
12/2023

CONTENTS

1.INTRODUCTION.....	7
1.1 Research questions	8
1.2 Methodology	8
1.3 Related studies	9
1.4 Structure of the thesis	10
2.CONTEXT.....	11
2.1 Information retrieval systems	11
2.2 Document management systems	12
2.3 Git.....	13
2.4 Gerrit.....	14
2.5 Markup languages.....	14
2.6 Functionality of a typical Search engine component.....	16
2.7 Microservices	19
2.8 Containerization	22
3.CURRENT SYSTEM AND PROBLEM STATEMENT.....	26
3.1 Markdown build pipeline.....	28
3.2 Current search feature	29
3.3 Problem statement.....	31
4.OBJECTIVES FOR A SOLUTION.....	33
4.1 Requirements for a new search feature.....	33
4.2 Selection of search engine component.....	37
5.DESIGN AND DEVELOPMENT.....	41
5.1 Interfacing the selected search engine, OpenSearch	41
5.2 Approaches to limiting the search results	42
5.3 Indexing	43
5.4 Functionality of the new search feature	45
5.5 New search UI	47
5.6 Tuning relevance scores	50
6.EVALUATION OF RESULTS AND FUTURE DEVELOPMENT.....	52
6.1 Effects on resource usage.....	52
6.2 Effect on speed	52
6.3 Estimated effect on security of the system	53
6.4 Future development	53
7.CONCLUSION.....	55
8.REFERENCES	56

ABSTRACT

Mika Hämäläinen : Integrating OpenSearch with a Git-Based Documentation System
Master of Science Thesis
Tampere University
Master's Degree Program in Automation Technology
December 2023

This master's thesis shows the process and results of improving the search feature of a document management system by choosing and integrating a search engine component into it. The document management system has a microservices-based architecture and is built on top of a version control system instead of a conventional database. The version control system has very strict controls on the documentation data being searched, and managing access control became important to the success of the project. These architectural choices caused additional challenges that needed to be addressed.

The thesis starts by giving some context about technologies that are relevant to the problem, and then its structure follows the design research method. First, the current system is introduced to give reader motivation and sense of the problem. Requirements for the solution are created based on the current architecture, features of the application, expectations about what new features users want and the constraints set by external systems and other stakeholders. These requirements include requirements for performance, security and new features that the users would likely want to have.

Based on these requirements, OpenSearch was chosen as the search engine component out of free, open-source alternatives. The selected component is analyzed and its integration into the document management system as a container is designed and implemented.

This integration was tested with end-to-end tests and is confirmed to satisfy the requirements.

Keywords: search engines, information retrieval, documentation systems, microservices

The originality of this thesis has been checked using the Turnitin Originality Check service.

TIIVISTELMÄ

Mika Hämäläinen : Integrating OpenSearch with a Git-Based Documentation System
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Joulukuu 2023

Tämä diplomityö esittelee prosessin ja tulokset projektista, jossa dokumentaatiojärjestelmää parannetaan valitsemalla ja integroimalla siihen uusi hakukone. Dokumentaatiojärjestelmä käyttää mikropalveluarkkitehtuuria ja on rakennettu versionhallintajärjestelmän päälle tavanomaisen tietokannan sijasta. Versionhallintajärjestelmä on asettanut tiukat rajoitukset haettavan tiedon käsittelemiselle, ja lukuoikeuksien hallinnasta tuli tärkeää projektin onnistumiselle. Tämä lisäsi uuden hakuominaisuuden suunnittelun monimutkaisuutta ja asetti sille rajoitteita.

Työ alkaa viitekehyksen esittelyllä: kertomalla teknologioista, jotka ovat oleellisia ongelman ymmärtämisen kannalta. Sen jälkeen sen rakenne noudattaa suunnittelututkimus-tutkimusmenetelmää. Ensin nykyinen järjestelmä esitellään, jotta lukija saa motivaation ja kuvan ongelmasta. Vaatimukset ratkaisulle luodaan järjestelmän nykyisten ominaisuuksien, nykyisen arkkitehtuurin, sekä sen perusteella, mitä ominaisuuksia käyttäjien ennakoidaan haluavan. Täytyi myös ottaa huomioon, mitä rajoituksia ulkoiset järjestelmät ja muut sidosryhmät asettavat hakukoneominaisuudelle. Vaatimuksissa on suorituskykyä, turvallisuutta ja uusia ominaisuuksia koskevia vaatimuksia.

Vaatimusten perusteella hakukonekomponentiksi valittiin, joukosta ilmaisia, avoimen lähdekoodin vaihtoehtoja, OpenSearch. Valittu hakukonekomponentti analysoitiin ja sen integrointi dokumentaatiojärjestelmään suunniteltiin ja toteutettiin, ja toteutuksen toimivuus testattiin.

Avainsanat: Hakukoneet, Tiedonhaku, Dokumentaatiojärjestelmät, Mikropalvelut

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

PREFACE

Journey up to this point has been longer than I originally expected. Finishing this thesis has taught me a lesson or two about not giving up. I would like to thank everyone who has helped me during my studies, especially my teachers, classmates, colleagues and my wife.

Tampere, 5th December 2023

Mika Hämäläinen

LIST OF ABBREVIATIONS

API	Application programming interface
BCPL	Basic Combined Programming Language
CORBA	Common Object Request Broker Architecture
CSV	Comma-Separated Values
DCOM	Distributed Component Object Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IP	Intellectual Property, basic unit of chip design
JSON	JavaScript Object Notation
LCX	Linux Container Runtime Project
MSA	Microservice architecture
QA	Quality Assurance
RMI	Remote Method Invocation
SGML	Standard Generalized Markup Language
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	System on Chip
SQL	Structured Query Language
UI	User interface
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language
YAML	Yet Another Markup Language

1. INTRODUCTION

The term “search engine” usually reminds reader of the familiar Internet search engines many of us use daily. Even the definition of the term “search engine” often includes mentions of the Internet. [8], [4]. Despite that, search engines are found in a wide variety of applications in which user must find a piece of information amongst collection of data that is too large to be manually searched. Some of them also perform optimizations meant to increase performance of the search compared to naively matching the search terms to the data being searched, and implement useful features, for example, Boolean queries. [2, features]. Nowadays there are also libraries available for developers who want to add a search engine into their own applications. [19]

Documentation Portal is an in-house, cloud-based, documentation retrieval and rendering application that was deployed in 2022. Its main user base are engineers working on creating integrated circuits. These projects, called SoC (*system on chip*) projects, can be very complex. SoC design is a methodology to help manage the complexity of developing large integrated circuits by way of better modularization and reuse of components. SoC projects often comprise multiple subsystems, each of which includes several IPs (*intellectual property*). IPs are the building block of an SoC project. They are designed to be reusable by rigorous testing and by having their own documentation in addition to the documentation of the SoC as a whole. As a side effect, there is potentially a big amount of version-controlled documentation related to a single SoC project. [29]

Documentation Portal enables a more efficient method of creating, distributing and storing all this documentation. With it, in-house projects can store their documentation in the same repository that they use to store their source code, and Documentation Portal can be used to search and retrieve the documentation from the repository and render it into an HTML (*Hypertext Markup Language*) page with uniform style across the company. By using the Documentation Portal, a developer can focus solely on creating content without having to spend time and effort on the style. The documentation is currently stored in Markdown format, but later there will be support for other types of documentation, including Doxygen. Documentation Portal is also able to generate graphs and diagrams without storing pictures in the repository, by rendering them from Markdown using free software called Kroki. [25]

Documentation Portal also has a simple search engine that allows users to search for documentation using a simple user interface. At this stage of Documentation Portal

project, the search engine is still passable, but it lacks many important features. Most notably, it doesn't offer a way to search inside the documentation contents. It is expected that the number of documents and users of Documentation Portal is going to rise significantly in the near future and a better search feature will likely be requested by users if not implemented beforehand. Decision has been made to improve this search feature by adding a search engine.

Objective of this master's thesis is to find solution to this problem of improving on the current limited search feature. This is done by establishing what the problems with the current implementation are and how the current Documentation Portal architecture works and using those to define requirements for a new solution. The new solution will consist of a new search engine and the necessary software additions to Documentation Portal that allow the new search engine to be integrated into it. The new search engine is selected from those that are available as a free software component. Shortlist of possible candidates is: Elasticsearch, OpenSearch, Apache Solr, Apache Lucene, Sphinx, Terrier and Xapian.

After selecting and integrating the engine, its configuration is studied. The UI (*user interface*) also needs to be modified to accommodate the new features of the new search engine while not removing features from the current user experience for those users who have gotten used to using the current version.

1.1 Research questions

Q1: What are the requirements for the implementation of the search engine component and what kind of architecture should be designed based on those requirements.

Q2: How to limit the access of users to only the documentation that they are allowed to see.

Q3: How is resource usage of the system affected by the change?

1.2 Methodology

This thesis uses the method called "design research". Design research includes six steps:

1. Problem identification and motivation. Motivate the research and audience of the research of why the problem needs to be solved and provide input for design of the design artifact that will be produced as a part of the research.

2. Define the objectives for a solution. Use the problem definition to create these. It should be stated how these objectives support solutions to the problems. If there are current solutions to the problems, their efficacy should be investigated.
3. Design and development. Creating the design artifact. The type of the design artifact could be any object in which the contribution of the research is embedded into the system. In context of software engineering, this could be the software that was designed.
4. Demonstration. This could be a simulation, case study or other proof of how the design artifact solves the identified problem.
5. Evaluation. Involves comparing the objectives of the solution to the actual performance of the artifact in step 4. This evaluation can be done in many different ways, depending on what metrics are available in the context of the research. Sometimes after this phase, the process is iterated back to design, but it is not required in all cases.
6. Communication. Communicate the solution to relevant parties.

[7 p. 28]

1.3 Related studies

Not much has been written about designing systems for crawling a private data repository, like Gerrit, for indexing documents into search engine index, compared to how much has been written about crawling the Internet and search optimization in Internet search context. These use cases for a search engine are not closely related to the objectives of this thesis, and the lessons learned in that field cannot be easily applied here.

One example of a study where search engine is selected and implemented for a private data repository is called *The Development of search engine service for official academic documents* by A A G Y Paramartha and L J E Dewi. It was made for Universitas Pendidikan Ganesha, Singaraja, Indonesia and the results published in journal called IConVET 2020, Journal of Physics: Conference Series. The paper documents how a search was created for official academic documents that are not publicly available. The researchers used Lucene-based search engine, Solr, and documented the process of indexing the documents in the source database, developing a query parser that maps the user queries into queries that directly input into Solr search engine and the evaluation of the search engine. They also used techniques called *field annotation mapping* and *n-*

gram mapping to increase the search performance and got positive results when evaluating their effects on the speed of the search. [1]

Another study, An Optimized Full-Text Retrieval System Based on Lucene in Oracle Database by Xiujin Shi and Zhenfeng Wang, published in Proceedings - 2nd International Conference on Enterprise Systems conference, uses Lucene and an Oracle database to implement and optimize a full-text search system that can search the database. The article documents the process of making the database data available for Lucene search and the multiple optimizations to the efficiency of the search. The results are evaluated with different test and while index creation is a lengthy process, especially with large amounts of data, the optimized index creation manages to gain some speed compared to the unoptimized one. [32]

There is also a study, called On the Auto-Tuning of Elastic-search based on Machine Learning, published in Conference on Control, Robotics and Intelligent System (CCRIS 2020) about boosting the performance of Elastic-search by using machine learning techniques. The study was done by Zhenyan Lu, Chao Chen, Jinhao Xin and Zhibin Yu from Shenzhen Institutes of Advanced Technology. The study began by measuring the performance of their Elasticsearch setup with no optimization in order to find some bottlenecks in its indexing and query operations and record the performance statistics with different configurations. Based on that data, a machine learning model, called Performance Predictor by the study, was created. A Configuration Optimizer was also created, which creates multiple configuration combinations and evaluates their performance with the Performance Predictor and selects the best configuration combination based on the results. They managed to reduce the operation delay on average by factor of 2.73 and up to factor of 7.02. [38]

1.4 Structure of the thesis

Chapter 2 goes through the background information about the technologies discussed later in the thesis. Chapter 3 Gives an overview of Documentation Portal application as it was before start of this thesis and presents the problem. Chapter 4 details the requirements created for the new search engine feature and the implementation based on those requirements. Chapter 5 Goes through the design and development of the new search feature. Chapter 6 Presents the evaluation of the results and future development plans. Chapter 7 Concludes the thesis.

2. CONTEXT

This chapter gives some background information on terms and technologies that are later mentioned in this thesis or are otherwise relevant to the Documentation Portal or the development of the new search engine feature. The chapter starts by describing information retrieval systems and document management system in general, to get viewer better big picture about the domain of the Documentation Portal.

After that the chapter goes through some of the technologies currently in use in Documentation Portal: Git and Gerrit form the external information storage where the searched data is located. When the new search feature is designed, it is important to know how they function, because the data is governed by these technologies.

Markup languages are introduced in this chapter as the last topic. This is important, because the data in Documentation Portal is in Markdown format, which is a markup language. It is important to know the format the data is in when designing a new search engine.

2.1 Information retrieval systems

Information retrieval systems are systems that store, organize and search information. The earliest form of information retrieval systems were ancient archives and libraries. Later automated systems were created, first using punch cards and microfilms as information storage media to manage growing amount of information produced by science and business.

Information retrieval systems might allow searching for the whole Internet or only a limited collection of files on user's personal system or anything in between. Some of them store and retrieve images, some videos, some only text, but they all have a shared goal: to allow user to select specific information that they want to have from a larger collection of information that they mostly don't want to have. [35 p. 15]

There are differences between different information retrieval systems, but on abstract level they can usually be broken into separate software components: One of them is called "indexing". The indexing part takes in the information stored in the system, collects relevant parts of it, and stores them in a collection that can be easily accessed. This allows fast matching of user queries to stored pieces of information. The other component is called "retrieval". It modifies raw user queries into form that can be used to score the search results based on the indexed information. [35 p. 15–17]

Modern examples of information retrieval system are digital libraries and web search engines. Documentation Portal also can be categorized as an information retrieval system, since it allows users to retrieve selected information from a larger collection of information, by using its search feature. Gerrit, from where Documentation Portal retrieves its data, could also be categorized as one, since it allows users to store, search and retrieve information and has its own UI that allows users to search for subset of the stored information. [13]

2.2 Document management systems

Before widespread digitalization undertaken by companies, documentation was often stored in filing cabinets as stacks of paper, and this practice is still being used in some places up to this day. Storing documentation this way requires room and it is easy to accidentally misplace papers. Storing and retrieving documents required lots of manual work when someone must physically shuffle through stacks of folders.

Solution to those problems is offered by document management systems, which are information systems that, at bare minimum, allow storage and retrieval of electronic documents. They solve the myriad of problems related to paper-based documentation systems and enable features that weren't possible before. Typical features that modern document management systems offer are:

- Advanced search: it is possible to search documents by using different search criteria.
- Version control: It is possible to view and edit multiple versions of documents and see who has made what changes to them.
- Security: It is possible to set different read and write access to documents for different users.

[5]

There are many different document management systems available. Some free, and some commercial products. Examples of popular documentation management systems today include Microsoft SharePoint and M-Files.

Git, which is used as a version control system in Gerrit, also offers above features. Git allows users to control access to documents they have created. It also features full version history, allowing tracking of changes made to each file by each user and rolling back to older versions.

2.3 Git

Git started as a version control system for developing Linux kernel in 2005. It was designed to be fully distributed and allow non-linear development with multiple parallel branches of development. Speed and simplicity were also prioritized in its design. Since its launch its user experience has been developing while still maintaining the simple design and speed that were the original goals of the project. [15 Chapter 1.2]

Major difference to most other version control systems is that Git doesn't store differences between versions. It stores full snapshots of the files if the files have changed between versions. Because of its distributed design, every operation doesn't need Internet connection to some centralized server, allowing many Git operations to complete fast. Especially when data is transferred over unreliable connections, data integrity checking is important. Git secures the integrity of files by having a 40-bit hash string for every file that changes when the files has been modified. [15 Chapter 1.3]

Git is the most popular advanced version control system, because it supports multiple different types of workflows and offers complex features that are often needed in complex software projects, such as branching, undo and merging, in a user-friendly way. It also has a large community that offers different choices for repository hosts and help with troubleshooting. In 2016, 70% of version control-related searches were about Git. [5]

2.3.1 Commit

Commit is a fundamental building block of how Git works. When user makes changes to files that are tracked by Git, the files could be added into staging area for a commit and then committed, saving the modifications in the repository that the user is currently working on. When this happens, Git stores a commit object that contains pointer to the snapshot of the content user had staged, author's name, author's email address and pointer, or pointers if there are multiple parent commits, to the commit that preceded the commit. User also writes a commit message that describes the changes made in the commit, which gets saved with the commit. [15 Chapter 3.1]

2.3.2 Branch

Branches make the non-linear way of working of Git possible. Every time a commit is made, a new snapshot of the state of the tracked files is created and a special pointer, called HEAD pointer moves forward to point to that new snapshot. By default, this happens in the default branch, which is usually called the "main" or "master" branch. When user has created a new branch and switched to that new branch with checkout-command

the HEAD pointer moves to point to the new branch. Now, when the user makes a new commit again, the pointer of the main branch will still point to the same commit, but the HEAD pointer and the pointer of the new branch will move to point to the new commit. If in this situation the user checks out the main branch and makes a commit in there, the branches will diverge and point to two different commits that are children of the same commit. Later they can be merged again and/or be branched into even more branches.

[15 Chapter 3.1]

2.3.3 Tag

Tags are pointers to specific commits. They are usually used to represent different versions of the files being tracked. They can be lightweight, containing only pointer to a commit, or annotated, which means they also show the tagger's information and an annotation message set by the tagger.

[15 Chapter 2.6]

2.4 Gerrit

Documentation Portal uses Gerrit server as its documentation storage. Gerrit is a Git server that manages standard Git repositories and provides users with additional features for managing them: It makes possible branch-level access control of Git repositories whereas Git only allows granting access to whole repository. It also has a code review feature. Standard installation of Gerrit doesn't provide many other features, like browsing and searching code, issue tracking, style checking, but it has a plugin interface that allows installation of plugins that offer many of those features.

There is no specialized Gerrit client and clients can communicate with Gerrit server using normal Git clients and Git commands, like push and pull. The code review feature is also optional and can be turned off for a branch or repository by allowing direct pushes. [14], [13]

2.5 Markup languages

Markup languages are text-encoding systems that are used to create documents that contain both the content text that is going to be displayed and control symbols, called markup. The markup describes how the text is supposed to be displayed when processed by the device that renders it. [10]

There are multiple markup languages in the world for different purposes. Some, like Markdown, YAML (*Yet Another Markup Language*) and JSON (*JavaScript Object*

Notation), are very popular and some, like SGML (*Standard Generalized Markup Language*) and Troff have been forgotten by many. Although the proponents of some format or other may argue strongly for using it in every place possible, the different formats often complement each other. When selecting which markup language to use, user needs to take into account the strengths and weaknesses of candidate languages and the contextual factors present in the problem at hand. Some things that should be considered before selecting the right markup language for a task:

- User needs to consider if the information needs to be made available in different languages with different alphabets.
- The different accessibility requirements of the readers should be considered.
- If there are relationships between documents, is support for those needed from the markup language, since some markup languages have features that support linking to other documents or even embedding other documents inside them.
- If the raw documents should be findable on the open web, the default formatting, in its unrendered form, should be readable by the search engines.
- Some markup languages offer support for validation if the task requires it.
- Life cycle of the documents: If the documents should be archived for a very long time, it would be good to have a feature in the language that documents the language and version used.
- Balance of offering very strong control for the developers and being too hard to learn or type: Very rich syntax can mean that it is possible to do everything that you want to do with the control symbols, but it might mean that the developers writing the documentation try to avoid doing it.

If the documents are going to be presented in the web and rendered by web browsers of the client users, the default format is HTML. It is very widely used because of that and the browsers can render even very old versions of HTML. [20]

Documentation Portal currently uses Markdown as the markup language with which the documents are created, but they are rendered into HTML for to be displayed in the browsers of the users. Standard Markdown doesn't include versioning information, but it is very easy to type and is readable even when not rendered into HTML. It also has support for links between documents. [20]

2.6 Functionality of a typical Search engine component

Nowadays there are many different search engine components that vary in their scope. Some are lightweight and only offer basic features, such as indexing and ranking of search results, while some are more heavyweight and implement huge selection of features for the user, sometimes even including the UI. There are some that are packaged as SaaS application, and some are just a library with some documentation on how to use it.

Common attribute of all these different search engines is that they all offer solution to the same core problem: indexing the data from some data source into an easily machine-readable format and ranking the searched content by relevance, based on user query and criteria that are specific to the use case or domain.

Most search engines follow the diagram pictured in figure (1). The search engine is connected to some data store, which is sometimes a local database, but in the case of Documentation Portal, it is an external version control system. For user, the search engine is a black box. If it's implemented and configured properly, the user doesn't need to know what happens inside it.

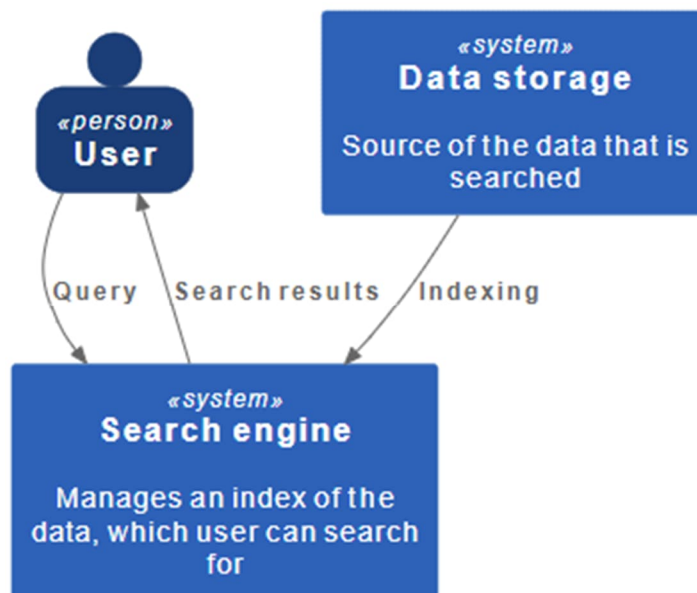


Figure 1: Basic principle of a search engine

2.6.1 Information collection and pre-processing

Before the data can be searched with a search engine, it must be made available to the search engine, by first exporting it from the original location, pictured as “Data storage” in figure (1). This original location could be a database or a collection of text files that the search engine can easily access, or it could be indefinite number of documents that are spread around the Internet, which is the case with Internet search engines. This raw data must be made available for the search engine and converted into a format that the search engine supports, called “document”. Documents are related to tables in SQL (*Structured Query Language*), because they also consist of fields that contain data that is stored as some datatype, for example String or Boolean. Sometimes the data source is structured in a way that the fields of the data source can be used as the document fields, but the source data often needs some modifications for search engine usage.

[28, Chapters 2.1.1–2.1.4, 2.3.1]

2.6.2 Inverted index

Indexing is the process of taking data from the data storage and using it to form the index that is used when searching for queried data. This data source could be anything that is machine readable: mp3 songs, images, books, Word documents. Search engine typically creates an inverted index out of it.

Inverted index consists of two parts: term dictionary and a postings list. Term dictionary is a mapping of all the terms that occur in a certain domain in a set of documents into numbered value. Its function is to give unique, short identifier to all search terms. Postings list contains mapping of search term identifiers to documents where they occur. Using these two, search engine can first find the unique identifier of the search term and then look for it in the postings list to find documents that have that search term. This explanation is simplified and many search engines add their own optimizations into this formula. [28]

2.6.3 Ranking search results

One very common feature for search engines is the ability to sort search results in different ways. One of those ways is to sort the documents by relevance.

Relevance quantifies how good a search result is considered by the user who conducts the search and it can be represented by a numerical value. This value is calculated by a function called ranking function. The ranking function can be implemented in many different ways to suit the user’s needs and possibly also the business objectives of the application. For example: Different fields can be given different weights and the results

where the search query matches with a heavily weighted field gets a high score and vice versa. A good ranking function probably also takes into account rarity of words and gives a better score when a rare word matches with a document. [28, Chapter 2.4.5]

2.6.4 Relevance tuning

Relevance tuning can be discussed in two contexts: One is altering a document (often a website) so that its relevance score is affected. Other context, which is used in this study, is when the search engine parameters are adjusted to provide better results for the users. Many search engines today allow the developer or administrator of the system to tune the parameters of how the search results are ranked. The default settings might not always be what the user wants to see from the results. Simple example: There is an index of books with fields name and description. One book, called “Forest”, has its name appear on the title field and not on the description field. Other one, called “Basics of orienteering”, might have “forest” in the description field multiple times. When user makes a search with a query “forest” with default parameters, where all fields are scored equally, the user might be surprised to see that the book he/she is looking for is not scored first. This is because the other book got multiple matches from its description field and therefore is ranked higher. This can be solved by tuning the parameters of the search engine. In this particular example, the field for the title of the book could be scored higher. This is called “boosting” in relevance tuning language.

One possibility to tune the search results is to filter the search results, based on what the user is expected to not want to see. The filtering is often a feature that the user can choose to use by using the UI of the search engine to set filters that exclude certain content from the results to make the results less cluttered. Depending on the business case and how the user experience is wanted to be tailored, filtering could also be done automatically on the service’s side when there is certainty about what the user is searching for. Tuning these kinds of filters might be a very complex task, like tuning the boosting of fields. [28 7.3]

There are multiple search engine specific books that include chapters on how to tune the relevance scores of the search engine. It is also a topic that big corporations are interested in, since patents have been filed and approved for methods of automatically tuning the relevance scores.

2.7 Microservices

Containerized applications have given rise to MSA:s (*microservices architecture*). There is a popular definition for term Microservices by Martin Fowler as *particular way of designing software applications as suites of independently deployable services* [22], [27].

Well before microservices, there were other technologies available for distributed computing, such as CORBA (*Common Object Request Broker Architecture*), DCOM (*Distributed Component Object Model*) and RMI (*Remote Method Invocation*). Emergence of these technologies was accelerated by the exponential growth of popularity of Internet-based services. They allowed distribution of applications over Internet in an object-oriented manner. They focused on delivering robust distributed applications that are tightly coupled despite being distributed. [11 p. 1–12]

Those technologies were the predecessors of SOA:s (*Service Oriented Architecture*). Using distributed applications, whose parts were developed by different teams, choosing to use different technologies, companies were having problems with islands of data getting isolated. For example, enterprise level decision making might require sales of two different departments to be readily available in their end, to be able to make data-based decisions, but each department have implemented their own systems. [11 p. 1–12]

SOA is a solution to this problem of making data integrated when it is distributed among a heterogenous system, by making the parts of the system more loosely coupled. The different parts of the system could be implemented using any language available if they just follow the communication protocol based on common message formats. messaging protocols. This could already be achieved with the technologies mentioned before (CORBA, DCOM and RMI) with their XML (*Extensible Markup Language*)-based communication formats, but the focus of those technologies wasn't data availability. [11 p. 1–12]

SOA made a breakthrough in popularity with the introduction of Web Services protocol. Web services works on top of HTML protocol and therefore is very firewall friendly. Older technologies can be integrated as web services into SOA by building them a web services wrapper. Web services stack also includes WSDL (*Web Service Description Language*) for describing the interface of the service with XML-based syntax, UDDI (*Universal Description, Discovery and Integration*) for publishing service details and the WSDL description. There is also SOAP (*Simple Object Access Protocol*) as a protocol

for invoking the services. All these technologies together enabled companies to integrate data in multiple different ways. [11 p. 12–20]

Two main categories of web services integration are point to point and central broker-based. The former category connects services to services using a connector component and doesn't scale well. The latter category uses a central broker where all the services are connected to. In its simpler form the services report their changes to the broker and the broker then notifies connected services about them. This enables looser coupling of services, but introduces a single point of failure into the system. More complex and intelligent version of broker-based web services integration is the ESB (*Enterprise Services Bus*). It has many advanced features. Few notable examples of those are security features, like authentication, routing services, automatic protocol conversions, for example XML to JSON and orchestrating services consisting of multiple smaller services. The ESB bus maintains a registry of all connected services using UDDI registry. Downside of the ESB is that its size increases in proportion to the number of connected services. [11 p. 22–24]

While SOA enables huge single application to be broken into multiple smaller applications that work together to provide a single service, microservices architectures work on a lower abstraction level than SOA to enable the same for a single application. It is not an alternative solution to SOA – they are both service-based, but can be used to complement each other. Figure (2) shows MSA and SOA working together. When using SOA, the single applications can be big, monolithic things, developed by 100 developers and deployed as a single file. MSA alleviates this by allowing these monoliths for be broken further apart into units that can be developed and deployed independent from each other. This is especially useful if there is need to deploy new features fast, from the request to deployment to the end user. MSA also allows for reuse of code, since the big monolith applications usually cannot be reused as easily as smaller services. [11 p. 30 – 32]

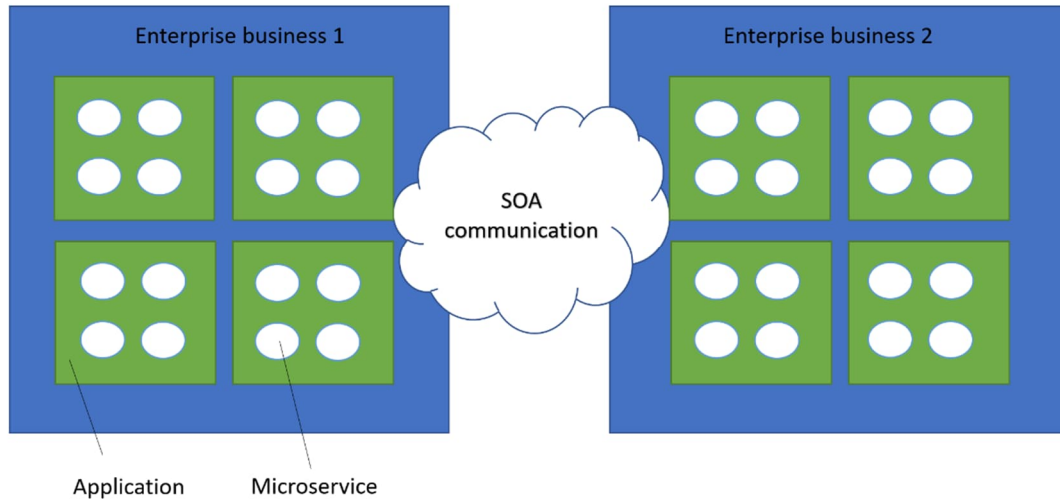


Figure 2: Business services connected by SOA using MSA.

Other advantages of MSA include better fault tolerance: If one service fails, it is isolated from the other services that might be able to continue operating without the failing one, and it is then known where to start looking for the reason for the failure. MSA also gives developers more freedom to choose what technology they want to use and helps avoid issue of vendor lock-ins. [11 p. 38]

Key enabler for the widespread use of MSA has been the availability of container platforms to run the microservices on and container orchestrators to help form bigger services from the individual microservices. [11 38–39] Especially popular one is Docker, which is used also in Documentation Portal. Another remarkable factor was the emergence of lightweight communication protocol called REST (*REpresentational State Transfer*), which works on top of HTTP. JSON message format also works well with REST to enable low overhead communications between microservices. [11 p. 43]

One of the most important disadvantages, that causes other disadvantages, is that MSA adds another layer of complexity. With the additional layer of complexity there is need for more complex DevOps (*Development Operations*) to maintain and deploy the services composed of multiple microservices. The number of possible combinations of configurations is very high when you consider together all the possible configurations the different microservices forming the bigger service have. While the freedom of using different technologies is also an advantage, it makes more possibilities for attackers to find vulnerabilities in the system. There is also documentation overhead, because each individual microservice should probably have its own documentation. And network and re-

source overhead for all the microservices communicating with each other instead of running in one monolith application. Testing and refactoring a complex system using MSA might also prove to be difficult. [MSA 45–46]

2.8 Containerization

Container is a package where the code is packaged with its libraries and other dependencies, such as environmental variables and configuration files, to form a unit that is transferable from one environment to another. It can be run on any operating system that supports the container platform the container was made for. Container platform can be lightweight, because it uses the services of the underlying operating system to provide standard running environment for the containerized applications no matter what operating system the container platform itself is running on. Multiple containerized applications can be running on the same operating system. Figure (3) shows how the container platform is running on top of the operating system and the containerized apps 1–4 are running on top of the container platform. These apps could be easily moved to a server with another operating system, which is also running the same container platform. Apps 5 and 6 are running normally on top of the operating system and they might not work if transferred on another operating system. [17]

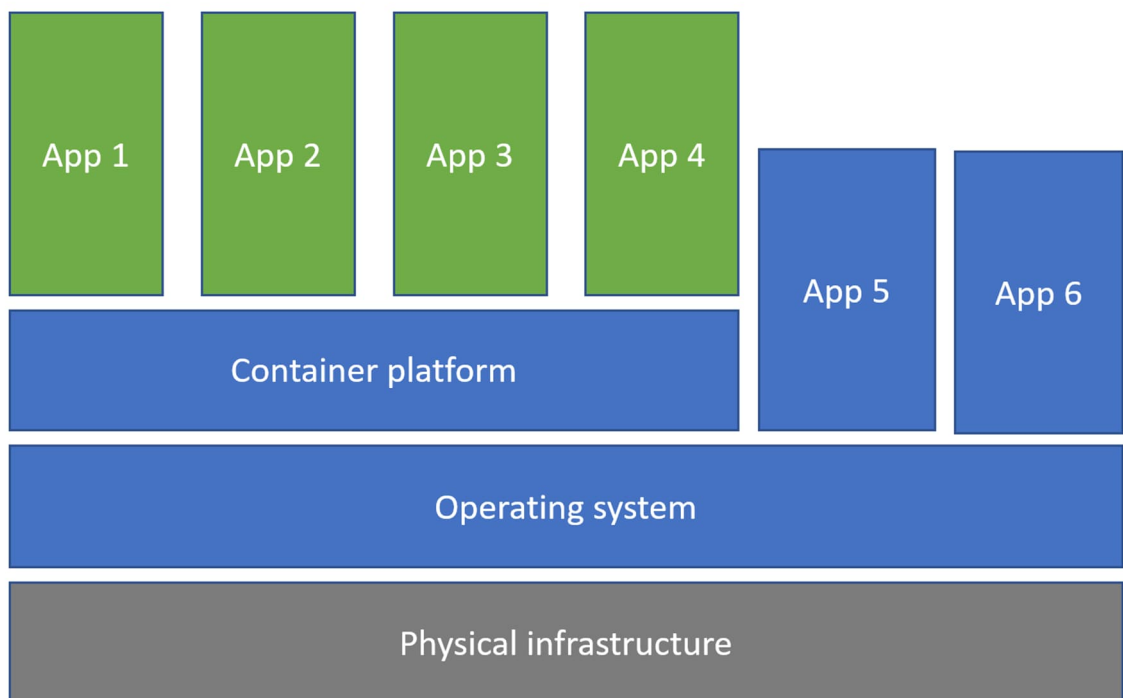


Figure 3: Diagram showing the architecture of a system using containerization

Another technique that can be used to solve the problem of getting application deployed on multiple different environments is a virtual machine. In this technique, the whole operating system is virtualized. Virtualization generally uses more resources than a container platform, because container platform can use the resources of the underlying operating system, whereas each virtual machine contains the virtualized hardware of the virtual host machine on which the operating system is running. [17]

History of containerization started with advantages such as control groups and namespaces that were invented during Linux development. Earliest form of this technology was *Linux Container Runtime Project* (LCX), which was a lightweight virtualization system. It was announced in 2008 and version 1.0.0 was released in 2014. Another important year was 2013, when Docker containers were released. There are alternative technologies for running containers, such as *Podman* and *Containerd*, but Docker has been the most popular out of these. [34] In addition to these container platforms, the invention of container orchestrators has been very important for the industry adoption of this type of lightweight virtualization, because they help with the problems running complex container architectures: They help with deploying, transferring and monitoring and allow easier scaling of containerized services, greatly reducing the manual labour associated with these tasks. There are multiple different container orchestrators in the market, but the most popular one is Kubernetes, which uses YAML-based configuration files for configuring the service.[17]

Containerization, especially using Docker, has become the default way to deploy applications for many developers. Sometimes it is used without consideration, even for simple applications that could be easily ran directly on the host machine. The popularity of this technology is explained by the multiple benefits it offers. Here are some of its benefits:

- Easier development: Containers make it easier for development teams to work on different projects inside the same bigger system independent of each other and create tests for their part independent of the surrounding environment. It also makes it possible to freeze the requirements. Many libraries develop fast and sometimes can introduce compatibility breaking changes. Keeping track of those changes requires maintenance time. Containerization allows isolating the application, or part of an application, and packaging it with its require-

ments, making it possible to use it even after the requirements have compatibility breaking changes, because every container can have its own versions of the libraries it depends on.

- Easier deployment: Containerization catalysed the rise of different architectures, such as microservices, that leverage the ease of deploying containerized applications. Because of general nature of containers, generalized continuous delivery and continuous integration systems could be created.
- Runtime benefits: Containers allow parts of the system to be independently restarted and updated. They also provide an abstraction layer between the operating system and the hardware that could increase security. They also make it easier to avoid creating interdependencies in data between different parts of the system, because the containerized applications often manage their own data.

These benefits come with some downsides, some of which are listed below:

- Design disadvantages: Containers promote blind reuse of choosing a seemingly applicable component for the system without considering what kind of legacy software it brings with its dependencies. The ability to develop the containers independently by different teams is also a double-edged sword. It might encourage teams to develop their containerized component without considering the big picture of how the component fits into the design philosophy of the architecture of the whole system. This might also cause problems when testing the whole system, in addition to the fact that relevant log files for debugging a problem might be hard to locate when doing system testing, composed of multiple containers.
- Deployment disadvantages: Development of container-based deployment strategies has shifted responsibility from traditional IT-departments to developers, who now need to spend more time on acquiring deployment-related skills and fixing deployment-related problems. One problem is also with deploying GUI (*graphical user interface*) applications with Docker. It wasn't originally designed to be able to run those, but there are workarounds that are cumbersome to use.
- Runtime disadvantages: While containerization is a lightweight way to virtualize a platform that app is running on, it also causes loss of performance. Especially when containerized apps are using different versions of libraries and they all need to be loaded into memory. There are also security concerns,

because container platform doesn't isolate the host machine from the running apps as well as a virtual machine does. There is also the issue of trust when the images are updated: should the developer blindly trust the new version of an image is not compromised? Monitoring a complex container-based system might also prove to be a difficult task. [34]

3. CURRENT SYSTEM AND PROBLEM STATEMENT

Developing embedded system-on-chip, SoC, projects is very complex and consist of many different IPs, developed by different teams who are managing their own documentation. Documentation Portal was created to solve problem with how SoC project documentation was being created, accessed and version controlled.

Documentation Portal is an in-house web application for searching, retrieving, and rendering documentation into static html pages with uniform look and feel. It was created mainly for SoC projects, for storing their documentation in the same repository as the source code but is also used for other documentation purposes, for example for storage and distribution of a technical manual.

Documentation Portal uses containerization with Docker containers and docker-compose for orchestration. Docker is widely used solution for running containerized applications. This means bundling all the application dependencies in one package, the container, which is then ran on Docker Engine. This allows application to be ran on any platform that can run the Docker Engine, which is available for multitude of platforms. This approach is a more lightweight way to package all the dependencies in one package than a virtual machine, because containers don't need to have virtualized operating system running on them. They are relying on the operating system on which the Docker Engine is running. [9]

Before Documentation Portal, documentation was mainly stored in various places, like SharePoint. Managing access rights and version history for these documents was sometimes tedious, manual work and finding them could provide to be a challenge when different teams had different practices for file locations. When updating a document, it might be necessary to also update a picture or a diagram and if the document is old, the original image or diagram might already be lost at that point.

When actual source files change, so should the documentation too, and when using older version of an IP, the documentation of the new version might not describe it accurately anymore. Therefore, it is advisable to find the version of documentation that matches with the version of the IP when using it. This was also harder before Documentation Portal.

By storing the documentation in Gerrit repositories with the code, problem with matching documentation versions with release versions is solved with Git tags: a tag

points to a certain snapshot of the repository where source and documentation should have a same version. Access control is handled by Gerrit and the users who have rights to view the code repository are also automatically granted right to view the documentation. Sometimes it is necessary to separate user documentation access rights from the codebase view rights, and in these cases a separate repository for documentation could be separated.

Currently the documentation is in Markdown format, but there are plans to expand Documentation Portal to support at least Doxygen format. Documentation Portal was designed to be expandable so that different build tools for converting different filetypes into HTML could be added.

Figure (4) shows how Documentation Portal operates on high abstraction level. The rectangles are all separate containers.

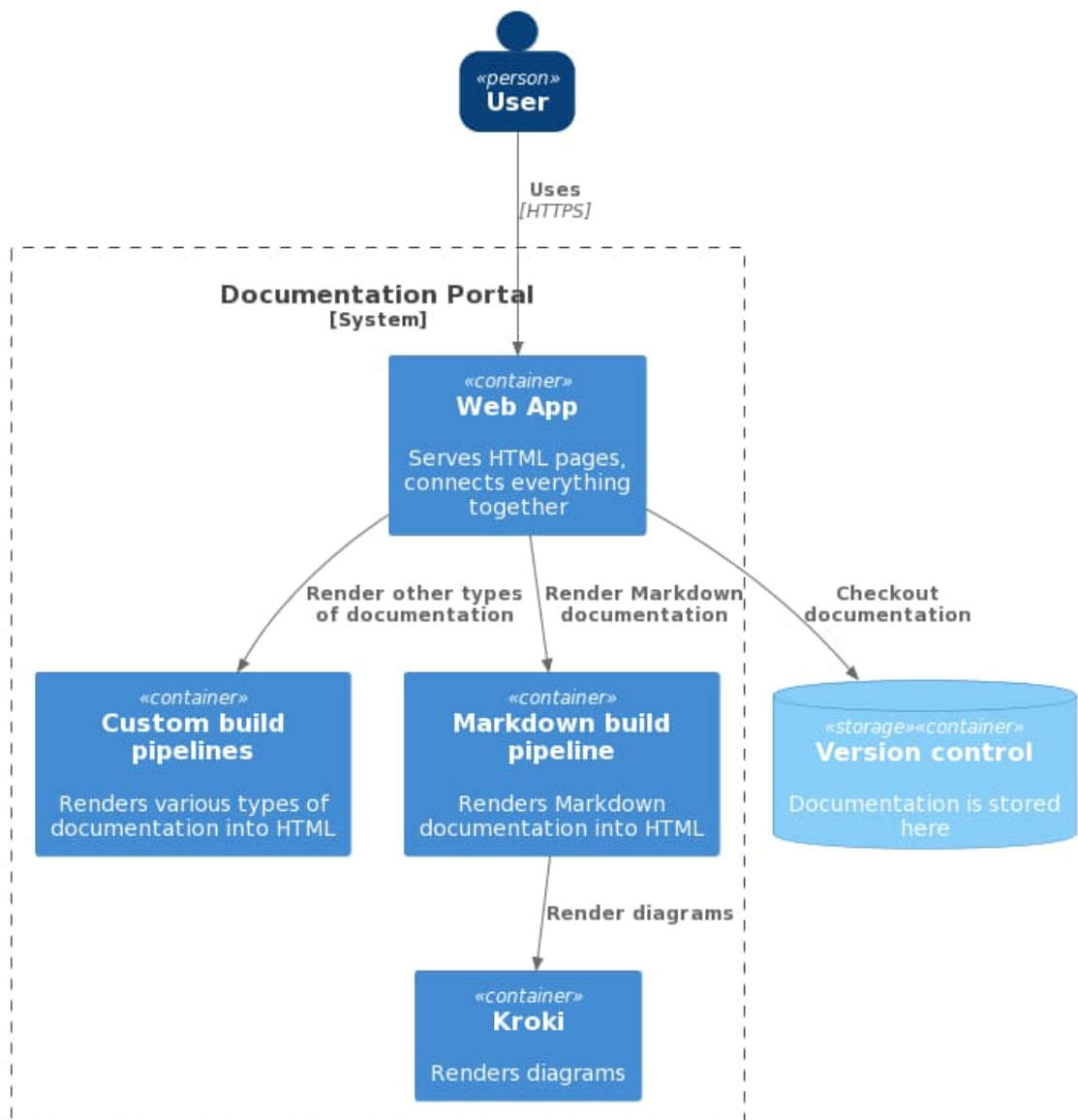


Figure 4 : Diagram showing how Documentation Portal operates on high level

3.1 Markdown build pipeline

Markdown build pipeline, shown in Figure (4), gets markdown files and a yaml (*yet another markup language*)-based configuration file as its inputs. Those are cloned from Git repository on a Gerrit server and they are fed as input to a container that returns a static html website, consisting of multiple html documents that are linked together. The container uses modified version of Markdown renderer called MkDocs.

MkDocs is a fast and simple generator, using Python Markdown library, for generating static html-pages from Markdown files. It's designed to be extended with themes and plugins and is open source, so the source code can be modified if the plugins and themes don't offer enough customization. MkDocs also offers a basic web server for quickly previewing the result when working with Markdown files, but many IDEs already have a feature to show a preview of rendered Markdown documentation. [23]

One of those is a separate Kroki container that is used to render graphs, diagrams and other graphical illustrations. These illustrations can be included in the Markdown documentation in textual form and are rendered as images that get added to the html document. [18]

When new build pipelines are added, they will be added as separate containers that the Web App can use for the parts of the documentation that cannot be rendered with the Markdown build container. Currently, only Markdown is supported, but Doxygen support is already being developed.

Markdown is a popular markup language that was designed to be as readable as possible. Working with Markdown documents differs from using WYSIWYG (*What You See Is What You Get*) language, because the formatting elements are added as text into the document when editing and their effect become visible when the document is rendered into, for example, pdf or html format.

The formatting elements are designed to be lightweight so that the raw Markdown files can be read and understood clearly. For more complex use cases, that are not covered by available addition, it's also possible to add inline html elements to a Markdown documentation.

Most Markdown applications implement their own version of Markdown. These versions are referred as flavours. Documentation Portal also implements its own flavour of Markdown with its mix of plugins and additions. Documentation Portal uses modified version of MkDocs -library for generating html pages from Markdown documentation.

3.2 Current search feature

Documentation Portal uses a simple search engine where user inputs query into an HTML form and it gets sent into Gerrit API (*Application Programming Interface*), that returns list of repositories that match the query. When user selects one of the projects in search results, a second selection tab shows all the branches and tags of the repository that can be filtered with a keyword.

This engine searches only matching text in repository names and every search creates a request for the Gerrit API. No content inside the repositories and documentation can be searched with it.

The behaviour of the old search engine is illustrated in figure (5). This is the sequence of events that happens in the use case where user is logged in with its credentials, searches for a document that is not cached yet, and opens in its browser.

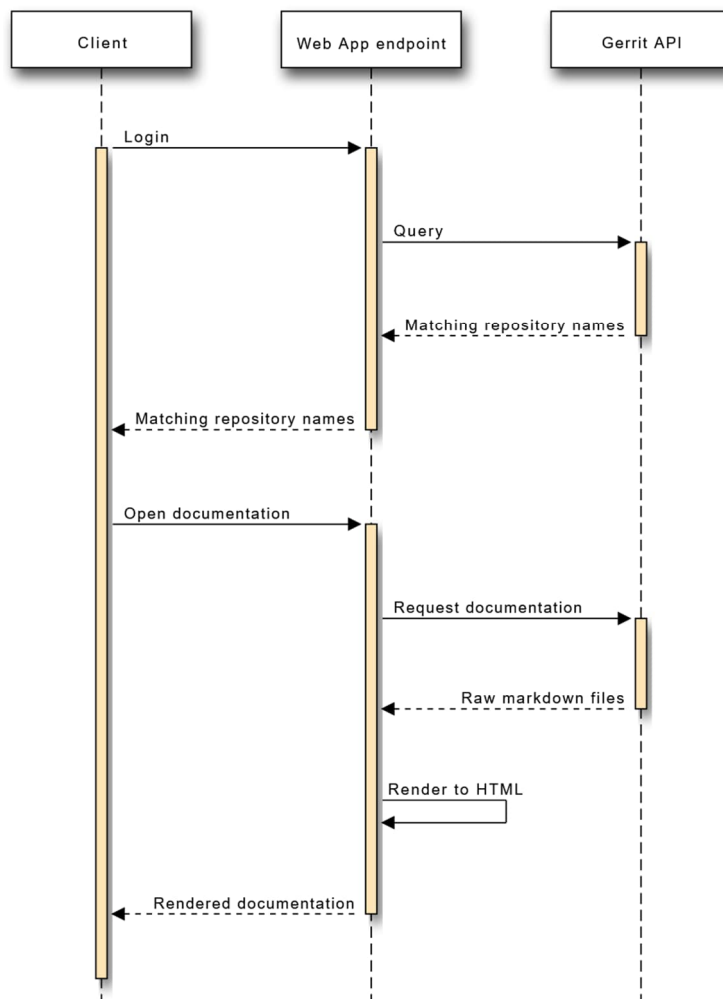


Figure 5 : Sequence diagram of how original search engine works

Actors in this sequence diagram of figure (5):

- **Client:** User using any browser, running client side Javascript code.
- **Web App:** Python code that implements the endpoints where client connects
- **Gerrit API:** Python Gerrit client that connects to Gerrit REST API.

Explanation of events of the sequence diagram from top to bottom:

In first part, user searches for the document he/she wants to open:

1. User has logged in with its credentials and submits a search query through a search
2. Query gets send, as it is, to Gerrit client, which communicates with Gerrit API.
3. List of Repository names that match the query are send back to Web App
4. User clicks on a repository name and list of branches and tags are requested from Web App
5. Web App requests the list of branches and list of tags, and other information, from Gerrit API
6. Gerrit API returns the information to Web App, which processed it into form that client can use

After user has selected the document, he/she wants to open, in this use case, he/she opens the documentation:

1. User clicks “open”-button of the Web UI. The Web App checks if the documentation is already found in the cache, does the version match with the one from Gerrit, and is the user able to open it.
2. In this case the document is not cached already, so it will be cloned from Gerrit
3. The files returned by Gerrit API are sent to render pipeline, which returns the html-site.
4. Web API caches the rendered files and returns the html site for the client, which opens it.

3.3 Problem statement

The current search engine is outdated and lacks features which users, who are familiar with other modern web applications, are accustomed to using. This decreases the overall user experience of the Documentation Portal, because it is too difficult to find

any document that the user doesn't have direct link to, if the user doesn't know its repository name. The search feature needs to be improved by having some of these features of a modern search engine, to improve the user experience. Most importantly, users should be able to search for the documentation content, not just the repository name.

The new features shouldn't be had at the cost of degrading other important aspects of the Documentation Portal service: Users still should not be able to access documentation that they are not allowed to access and the speed should not be affected negatively in a way that disturbs users.

4. OBJECTIVES FOR A SOLUTION

It was decided to improve the search feature by creating a new search feature around a search engine software component. First step in creating this solution is creating requirements for the new search feature and then selecting a search engine component, using the requirements for the solution as a guide to rule out the candidate technologies that are not compatible with the requirements. After the selection of search engine component, an architecture that integrates it as a part of the Documentation Portal is designed. This integrating architecture must take into account both the objectives for the new search engine feature and the requirements that the search engine component has on the system: the interfaces it can be connected with and the format it needs the data to be pre-processed to before indexing.

4.1 Requirements for a new search feature

The current search engine is used as a basis for creating requirements for a design artifact that is intended to improve it. In this section those requirements are created, based on developer intuition and feedback from small subset of users. As designing a search engine feature is often iterative process, these requirements are subject to change even after this study is published.

Some of these requirements affect the selected architecture for the solution, some the selected search engine software component, and some affect both. All the requirements created for the design are listed in table (1) below and their effects below the table.

Table 1: Requirements for the new search feature

Description	Pri- ority	ID
The search engine must be possible to run locally.	1	req 1
No data is leaked to 3 rd party servers.	1	req 2

Users shouldn't be able to access material that they don't have permission to see. This includes parts of documentation stored in the system.	1	req 3
Speed must not be noticeable slower than of the current search feature.	1	req 4
User experience of users who like how the current search works should not be negatively affected.	1	req 5
Ranking of search results	2	req 6
Developer should be able to finetune the way search results are scored and ordered.	2	req 7
The search engine should be able to produce results, based on contents of the documentation, not just metainformation.	2	req 8
Free use	2	req 9
Open source	2	req 10
Can be run in a container	2	req 11
Search keywords NOT, AND, OR	3	req 12
Phrase search	3	req 13

Requirement 1: The search engine must be possible to run locally. The data should be only stored on in-house servers.

Effects on design: When selecting a component, the ones that must be run in external cloud should be excluded from the search.

Requirement 2: No data is leaked to 3rd party servers.

Effects on design: Proper security measures must be taken to avoid outside access or unauthorized access to the system. Open-source components that are actively developed and are prioritized when selecting the search engine component, because it is less likely that they contain features which leak information outside. Older component with only few users might have some security issues which have been noticed a long time ago, but are not fixed because of lack of developer contributions. With closed source components, it is not as easy to tell what they are doing with the data, because only the developers have access to the source code.

Requirement 3: Users shouldn't be able to access material that they don't have permission to see. This includes parts of documentation stored in the system.

Effects on design: Since the access rights are stored in Gerrit, there should be some way to design the system to sync the access rights from Gerrit. This could be done every time the user makes a search query, avoiding the issue of having to store redundant access rights information on the Documentation Portal server. Other option is to store the access rights information on the Documentation Portal server and sync it with Gerrit periodically, making a more complex design, but likely faster search experience.

Requirement 4: Speed must not be noticeable slower than of the current search feature.

Effects on design. Since Gerrit API is the bottleneck, minimum number of calls to it should be made. The indexing process should be made so that the user doesn't experience slowdown of the service: The indexing should be done in its own thread, process or even as a separate microservice, if needed.

Requirement 5: User experience of users who like how the current search works should not be negatively affected.

Effects on design: This is a standard feature for modern search engine components. This requirement is satisfied with any of the candidate search engine components.

Requirement 6: Ranking of search results

Effects on design: This is a standard feature for modern search engine components. This requirement is satisfied with any of the candidate search engine components

Requirement 7: Developer should be able to finetune the way search results are scored and ordered.

Effects on design: Being able to set weights for different fields in the index is a very common feature of modern search engine components.

Requirement 8: The search engine should be able to produce results, based on contents of the documentation, not just metainformation like tags and title.

Effects on design: This requires for full text of the documentation to be indexed, making the indexing process little bit heavier, since the whole documentation needs to be checked out of the repository for indexing to be possible. This problem is alleviated by the fact that the documentation is in Markdown format and therefore only text needs to be indexed.

Requirement 9: Free use

Effects on design: This affects the selection of search engine component, but there are multiple modern free to use search engine components available.

Requirement 10: Open source

Effects on design: This affects the selection of search engine component, but there are multiple modern open source search engine components available.

Requirement 11: Can be run in container

Effects on design: This affects the selection of search engine component, but there are multiple modern search engine components available that can be run in a container.

Requirement 12: Search keywords NOT, AND, OR

Effects on design: Boolean search is very common feature with modern search engine components, but this must be made possible by the UI. If the user has to use some search engine specific syntax, it must be documented clearly or the UI should give some tools to make its use easy without having to learn the unfamiliar syntax.

Requirement 13: Phrase search

Effects on design. This is also very common feature of modern search engines. The web search engines that are very common tool nowadays use double quotes when searching phrases. If the selected search engine component uses some other syntax, it should be made easier by the UI and documentation.

4.2 Selection of search engine component

This sub-chapter goes through the candidate search engine components and gives some details about their attributes and differences. The search engine component that is used to implement the new design feature is selected from the candidates. Only search engine components considered further are ones that are released as open source with a permissive license, such as Apache or MIT license. This leaves out some popular search engine components. The candidates that were considered are listed on table (2) below.

ElasticSearch is one of the most popular search engines, but its license was changed from Apache 2.0 license to a dual license, where user can choose to use either Elastic license or Server-Side Public License. Elastic license doesn't allow user to provide the software to a third party as a hosted or managed service and Server-Side Public License makes it necessary to provide all the source code of the service using the licensed software as open source. Both constraints might become problematic for the future of Documentation Portal application.

Table 2: search engines compared

	OpenSearch	Apache Solr	Sphinx	Terrier	Xapian
License	Apache 2.0	Apache 2.0	GPL version 2	Mozilla Public License 1.1	GPLv2+
Updated in 2023	yes	yes	yes	no	yes
Popularity [6]	4.	3.	8.	< 24.	N/A
Implementation language	Java	Java	C++	Java	C++

Above table (2) shows the main attributes of the search engines in this comparison.

The popularity is from DB-Engines.com -website, which ranks, amongst other technologies, search engines, based on their popularity. The ranking standings are from August 2023 standings and the rankings are based on the proprietary scoring formula of the website.

4.2.1 Review of candidate components

In this chapter all the candidate components briefly introduced and their main features and differences from each other are explained. The chapter doesn't go into small technical details of their inner workings, but gives an overview of the technologies instead.

OpenSearch

The company behind Elasticsearch, Elastic NV, announced in January of 2021 that they would move away from the Apache License and start using licenses that offer less freedom for the user. Amazon AWS was offering Elasticsearch as a SaaS (*software as a service*) and decided to fork the project, to avoid vendor lock-in, giving start to OpenSearch. [3]

Like Elasticsearch, Opensearch is based on Lucene. Under Lucene there is lucene core, which is a search engine library implemented fully with Java that is used as a base in many other projects related to search, information retrieval and analysis. It claims to have high-performance and small memory requirements. It also offers many features that are relevant for a modern search engine: Ranking and sorting of search results, search fields (eg. author, contents), suggestions and typo correction, extensible ranking models and more. It is available under Apache License 2.0. [26],[2]

Opensearch is designed to be easily scalable by being distributed into clusters. User can have one or many of them and they can be distributed onto multiple physical or virtual machines. Data is stored in indexes that in JSON format and the indexes are stored in internal structures called shards. One shard is a full Lucene index and splitting an Opensearch index into multiple shards created redundancy that allows the search service to continue functioning in case a computer malfunctions. [26]

OpenSearch extends Lucene in multiple ways. The ranking algorithm is BM25, which is based on Lucene's TF/IDF framework. [26]. The BM25 algorithm is very widely used and successful algorithm. It has been in development for 30 years and is in use for many web searching applications and other business applications [30].The algorithm can be fine-tuned for specific problems by adjusting its many free parameters, but this is often not worth the time investment, since human is required to analyse the results of the optimization by evaluating results of multiple queries. [31]

Apache Solr

Apache Solr is also based on Lucene. It was developed as a side project to Lucene first but starting 2021 managed as a separate top project. Since OpenSearch is also based on Lucene, most of the things said about it also apply to Solr. Solr differs from OpenSearch. One of the main differences is in the data schema definitions: Solr offers

more flexible way to define what data goes into the index and what kind of search results are returned: In OpenSearch user defines the fields and their datatypes that the documents in the index will have, but in Solr new fields can be added on the fly. Solr also offers more flexibility with how the data is imported and exported, because OpenSearch works with JSON interface, but like Solr, it can also use xml and csv (*Comma-Separated Values*) files.

When it comes to speed, some comparisons of Elasticsearch against Solr have been made, which indicate that Elasticsearch would be faster if indexing and searching is happening concurrently, [21], [12]. These are old and they don't compare Solr with OpenSearch, but OpenSearch and Elasticsearch share many similarities, so the results can be considered when choosing a search engine.

Sphinx

Sphinx is a search engine written in C++ and distributed under GPL license version 2. It is a standalone software package that can be integrated into many different systems, because it offers multiple interfaces: API is available for multiple languages with the official distribution and for even more as third-party add-ons. In addition to having multiple APIs, it offers direct support for integrating with MySQL and PostgreSQL databases. [31]

Sphinx website has claims about how fast the search engine is. Indexing speed of 10-15 MB/second and 150-250 queries/second for every core in 1,000,000 document index. The website also lists many features for the search, that are commonly found on popular, modern search engines, like Boolean search, different relevance ranking algorithms and query suggestions. There is also possibility of setting up a multi-server system and getting federated search results from multiple servers. [31]

Terrier

Terrier is a lightweight open-source search engine developed by the Information Retrieval Group of School of Computing Science at the University of Glasgow for the purpose of research in text retrieval. It is developed in Java programming language and work across various different operating systems supporting Java.

For research purposes, it supports batch indexing of various standard test collections and changing of the ranking function is also supported. There are multiple ranking functions available to use with it, including one with learning capabilities. Other features mentioned on the website include support for machine learning models, modular and extensible indexing and querying APIs. It has its own query language that supports advanced features such as synonyms, proximity search and +/- operators. [33]

Xapian

Xapian is a search engine that was originally based on Muscat information retrieval system, that was written by Dr. Martin Porter at Cambridge University in a programming language called BCPL (*Basic Combined Programming Language*). Rights Muscat were bought and sold between multiple companies and then acquired by BrightStation PLC, which started to develop it closed-source under a different name. The last public version was used as a base for a new project under GPL license that became Xapian, which is now using GPL v2+ as its license. Unlike original Muscat, Xapian is written in C++ with bindings for multiple other popular languages. It features support for multiple ranking models and rich selection of boolean query operators. [37]

5. DESIGN AND DEVELOPMENT

Integrating the search engine component consists of 2 major problems that are intertwined. One of them is creating the indexes: Deciding what data to index and what kind of fields the documents in the index should have. This will affect also what users will be able to search for. Other one is providing a good and safe user experience for the user: Displaying the search results that are relevant to the user and not displaying any results that the user should not be able to see, based on his/her rights on Gerrit repositories.

5.1 Interfacing the selected search engine, OpenSearch

Out of the candidate search engines, OpenSearch was selected. There were other good choices, but a prototype was first developed with OpenSearch and its interface proved very good to use. OpenSearch was considered first, because it is very popular, actively developed and has favorable license terms. These also satisfy requirements 9 and 10. It can also be run locally as a separate container, satisfying requirements 1 and 11.

When it comes to features, OpenSearch, as many other considered candidate search engines, satisfy requirements 6, 7, 8, 12 and 13. It also has many other features that are not even taken advantage of when designing this search feature, but be put to use in later development.

OpenSearch also has a Python client available, which will be used to connect rest of the application with it, since backend of Documentation Portal is written in Python. The interface uses JSON as its communication format, which translates easily into Python dictionary datatype. [26]

When creating an OpenSearch index, the format of the data needs to be decided. The data should be split into different fields, which can be then searched separately if needed and given different priorities. The data that is sent to OpenSearch as JSON object, created by the Python client from Python dictionary, must follow the same structure that the index has. Multiple indexes could be created if there is need for multiple types of data.

For Documentation Portal, only one index is created, which has the following fields:

- “repo_short”: The short name for the repository. It doesn’t include the full name of the repository, where different levels of hierarchy are separated by forward slash. Only the last part is taken for this field, because it can be considered as the repository name, when talking about the repository and therefore something that the user might instinctively try to search for when looking for a certain repository.
- “repository”: The full name of the repository.
- “ref”: The name of the branch or tag of the document. It was decided to index documentation from latest commit of every branch and every tag of the repositories that have documentation. This field will not be used in search, but with the repository-field it can be used to identify documents uniquely.
- “author”: Name and email address of the author. They can be added to a list with 2 items, so the user can search for either name or email address when needed.
- “tags”: placeholder for an upcoming feature, if there are tags that help advertise the content of the documentation
- “revision”: The unique revision hash that Git gives for each commit. This can be used to check if the information in Gerrit repository has been updated since the documentation was indexed and therefore in need of updating. This will not be used in search.

5.2 Approaches to limiting the search results

Requirement 5 mandates that users should not be able to see search results that they are not allowed to access. This access control information comes from Gerrit. Two approaches were originally considered: Creating a separate index for every user and creating a common index for all the users. The latter approach would allow fast search results that are limited to show only results that the user can see, because the current user would be only allowed to search from a personal index, that only contains the documents he/she is allowed to access and therefore a call to the Gerrit API would not be needed to get the access control information before every search to filter the results. Downsides of this approach are much bigger memory requirements and new users not having any search results available when first accessing the app. The memory requirements would be high because many documents would be stored multiple times in different repositories and there and the personalized index would need to be created for every new user when they log in to see any search results. A situation is also possible where user’s rights to a

certain repository would be revoked when the documentation from that repository is already indexed and the search engine would not automatically get any warning when that happens. This would allow the user to get access to information that he/she shouldn't get access to, unless the rights would be checked from Gerrit when the document is requested by the user, and that would negate the speed gain of this approach.

With one combined index approach all the documents are indexed into the same index. This allows new users to have some search results readily available when first accessing the application if they share rights to some documentation repositories with other users.

5.3 Indexing

This sub-chapter describes how the Documentation Portal indexes documents into the OpenSearch container. There is a problem that contents of Gerrit, including documentation, are in constant change and Documentation Portal cannot be notified of these changes. Therefore, the index needs to be updated constantly, by comparing the Gerrit repositories containing documentation with the documents indexed in the OpenSearch index.

Figure (6), below, shows how the indexing process works when a user logs into the application. First the web app creates a GerritScraper instance, which contains a Gerrit client instance, using the user's credentials.

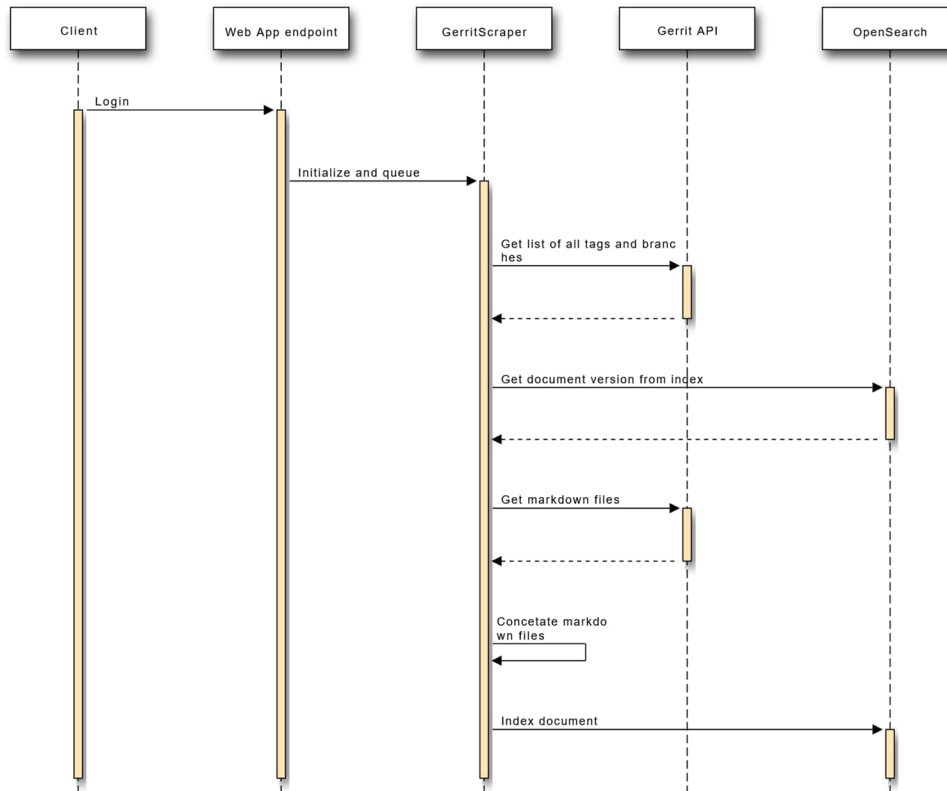


Figure 6: Indexing process

Actors in the sequence diagram of figure (6):

- **Client:** User using any browser, running client side Javascript code.
- **Web App:** Python code that implements the endpoints where client connects
- **Gerrit API:** Python Gerrit client that connects to Gerrit REST API.
- **GerritScraper:** Python class that is responsible for scraping the Gerrit repositories from any documentation that can be indexed. It contains a Gerrit client, with user's credentials, allowing it to access the documents that the user would be able to access in Gerrit. It inherits from Thread, to make it possible to run multiple concurrent scraping threads. Concurrency is managed by ThreadQueue class that manages which GerritScrapers are running and which are waiting for their turn, to limit number of concurrent actions.
- **OpenSearch:** OpenSearch service running in its own container. Access to it is managed by SearchManager class, which implements the search and indexing functions, using low level OpenSearch Python client library.

Explanation of events of the sequence diagram from top to bottom:

1. Client logs into the application
2. Web App endpoint authorizes the login and creates a GerritScraper instance, which contains a Gerrit client that can use the user's credentials. The GerritScraper instance is put into queue and gets run when the amount of scrapers running is lower than the maximum
3. GerritScraper uses Gerrit client to get list of all branches and tags that the user can read from Gerrit and which have already been rendered by Documentation Portal at least once. This can be checked by testing if the documentation is in the documentation cache of the application.
4. Web App searches the OpenSearch index for all of these documents and compares their revision hash with the ones from Gerrit. If the one from Gerrit matches with the one from index, the document is skipped. If the document is found in the index, but revision hash is different, the document is updated. If the document is not found in the index, it will be indexed there.
5. The indexing/updating process: First the markdown files are cloned from the repository and concatenated into one string. JSON document is created where other fields are taken from the previous Gerrit client response and the text_body-field is the concatenated string of all Markdown files in the repository. This is then indexed, or if there is an older version in the index, the version in the index is updated, using the SearchManager. There is a possible race condition that might lead to a document being indexed twice: If two threads start to process the same document and one of them checks that the document is not indexed and prepares to index the document, while another thread comes to the same conclusion and then they both index the same document. Therefore, this part of the GerritScraper code is protected by a lock that allows only one thread run the protected code at a time.

5.4 Functionality of the new search feature

It was decided that the old search feature needs to be preserved for those users that still want to continue using it, and for those situations where the documents that the user needs are not indexed yet, and therefore cannot be found using the new search feature. Some users might also enjoy the possibility to search for all repositories and not only the ones that have Markdown documentation. The new search engine is activated by selecting a checkbox that indicates if the index for the user is still being built, is being updated,

or is finished. This allows user to know how up to date the information, that he/she will get from the search results, is.

There are multiple ways to search in OpenSearch, but for this application, query string is selected This is because it allows some features that are minor requirements for the search engine.

Figure (7), below, shows the sequence of events when user, that has logged in, searches for a document and selects one of those documents, out of search results, to be rendered and opened.

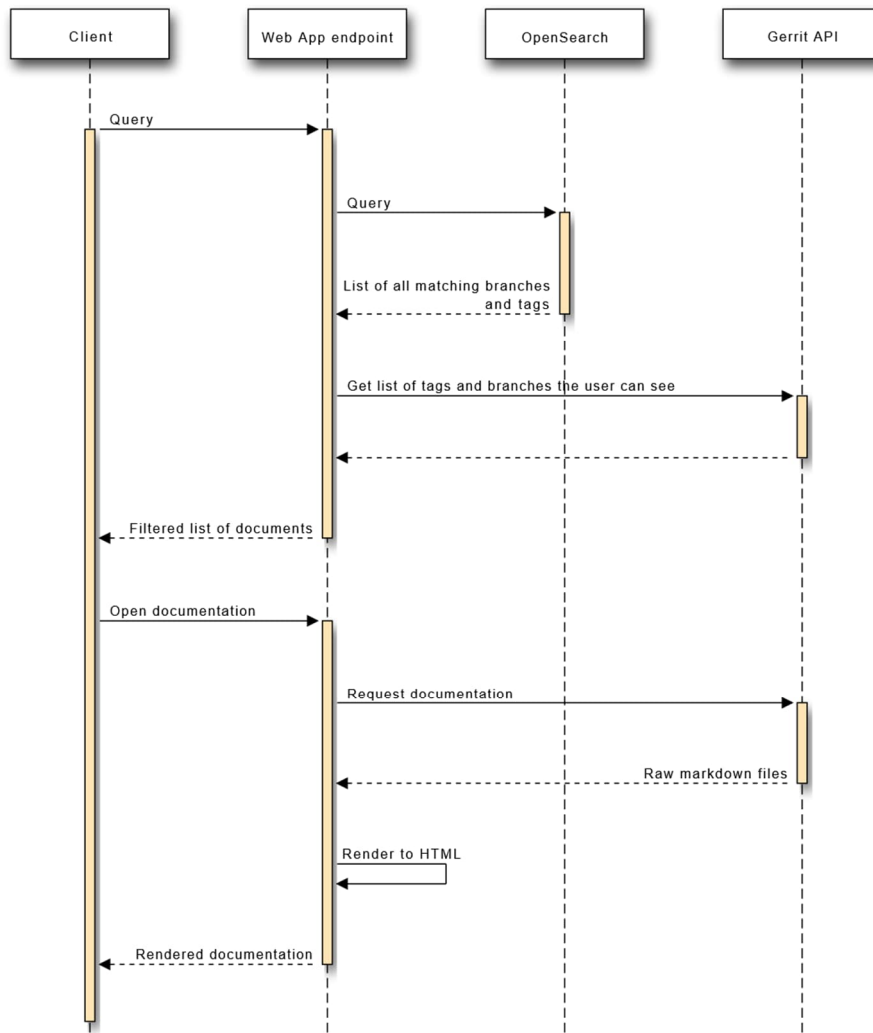


Figure 7: sequence diagram of user searching and opening a document with the new search

Explanation of events of the sequence diagram of figure (7), from top to bottom:

- User inputs the his/her query string into the search text box of his/her browser and presses enter or clicks the search-button. This sends a request to the Web App endpoint.
- Web App endpoint sends the same query into OpenSearch, through Search-Manager-class. The search query is only slightly altered, to remove special symbols which would cause error with the OpenSearch container.
- OpenSearch container returns results that contain the name of the repository, the name of the branch or tag and the relevance score.
- Web App requests a list of tags and branches from Gerrit API from every repository that was in the OpenSearch search results and filters away the search results that are not in the response from Gerrit, because user is not allowed to see those.
- The filtered list of documents is sent back into client, sorted by relevance score. The results are displayed for the user and the search result is saved on the client's memory until the next search is made.

5.5 New search UI

The new UI was constructed so that the old search functionality is the default option, although that might be changed later. In the old UI, when user does a query, it gives all the results whose name matches with the query string on the left section of the view, labeled "Search Project". These results can include also repositories that don't have any documentation and there is no indication which ones have documentation. User then must click a repository to see list of tags and branches inside the repository in the "select version" section. After clicking on a branch or tag in that section, user sees some information about the project and the branch/tag on the right part of the view in the section that is labeled "Please select a project to view details" in figure (8). User must go through all these steps to see if there is documentation that can be opened in the repository.

The new search feature can be activated by clicking the checkbox with label "Markdown search", next to the search bar, visible in figure (8). This empties the search field and the results and then the new search feature is activated. After the new search feature is activated, it can be deactivated by clicking the same checkbox

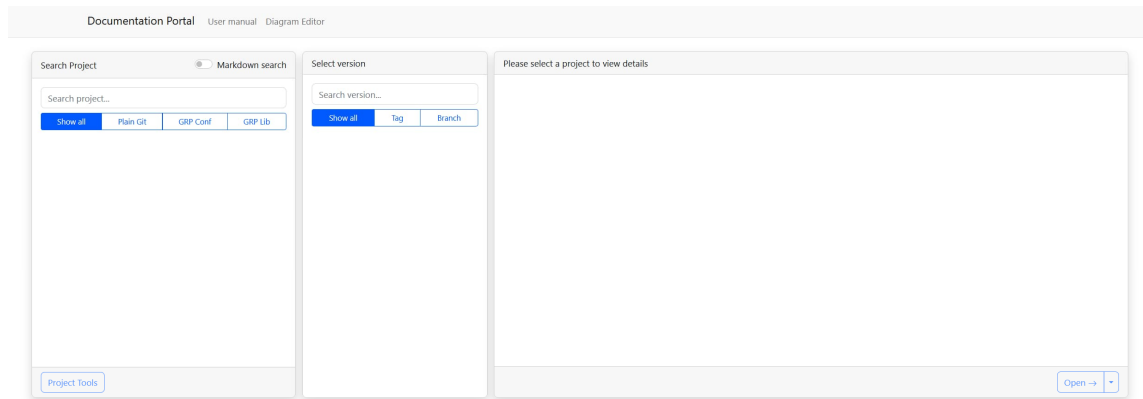


Figure 8: New search feature deactivated

One notable difference about the new search feature is that when user does a query, it only shows repositories that match the query, and all of those results have documentation that can be opened by the user. This makes it much easier for the user to find what they are looking for. The results are arranged in order of relevance score, which is the maximum relevance score of the branches and tags of the repository. When user clicks on one the repositories in the left section, the list of branches and tags is opened, similar to the old search. With the new search, all the branches and tags are displayed, ranked by relevance score, but the ones that don't match the search are shown with relevance score of 0. This decision to show the branches and tags that don't match the search was made after tester feedback on the prototype version.

Figure (9) shows how the new search feature looks when a search with query “documentation” has been made. The names of the repositories, tags and branches are anonymized.

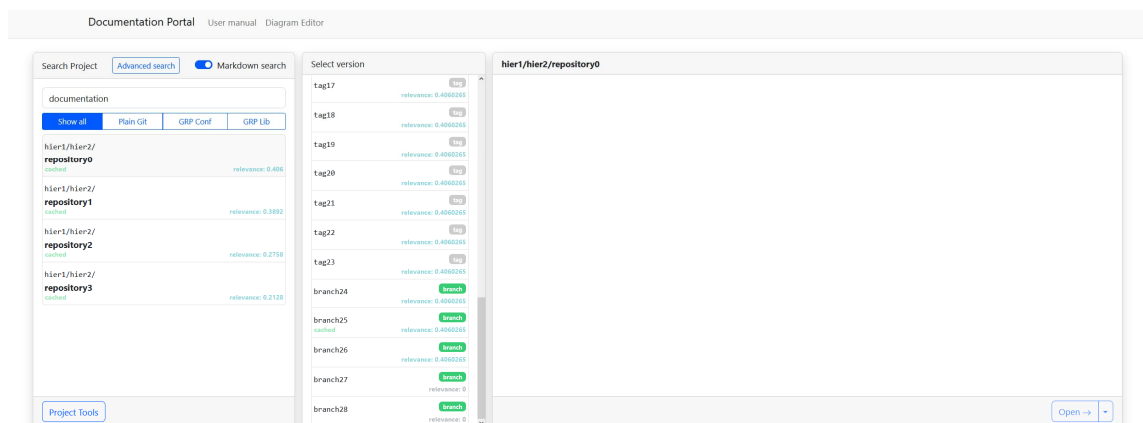


Figure 9 : New search feature activated with a repository selected

By clicking on the button that says “Advanced search”, user can open the UI that can help him/her construct more complex queries. The search field already accepts Lucene query syntax, but it was decided that it would be better to give something for the

users that don't want to learn it, but still want to make bit more complex queries. The query constructor that opens from that button is shown in figure (10) with multiple lines added. Using the "Advanced search" UI, user can create and execute a query that searches selected fields for the search terms or excludes results that have the terms in selected fields.

Meaning of choices in the dropdown, that is currently opened in figure (10):

- "any field": all fields are searched for the search term
- "document content": only the body text of the documentation is searched for the search term
- "repo name" only the repository name, without the hierarchical path, is searched for the search term
- "repo full path" only the full hierarchical path of the repository is searched for the search term, which is useful if the user wants to search for all projects under a certain organizational level.
- "author" searches for both the author's name and email address.
- "description" searches for the repository description, which was also indexed from Gerrit, with every document.

The search term can be also excluded from the search results by selecting "cannot contain" from the dropdown that now reads "Must contain". For example, if the user doesn't want to see matches from certain author, he/she should add a search term where "author" is selected from the first dropdown, "cannot contain" from the second and write the excluded author's email or name in the "search term" textbox.

The "fuzzy search" checkbox allows user to activate fuzzy search feature for the search term. This is an OpenSearch feature that causes words that closely resemble the query to also match the query. It is helpful if the user makes a typo when writing the word.

Pressing the "Search" button causes a query to be written in Lucene syntax that is executed in the main window, just as the user would have executed a similar query. The query is left visible to the user, in case the user wants to modify the query in the textual form. The user can also choose to open the Advanced search window again and the information he/she entered is saved from the last query made.

If the user input a search term that consists of multiple parts, the query created automatically puts it into quotes as a phrase. Then the “fuzzy search” checkbox is disabled, because it doesn’t work with phrase searches. Low priority requirement 13 is also satisfied with this feature.

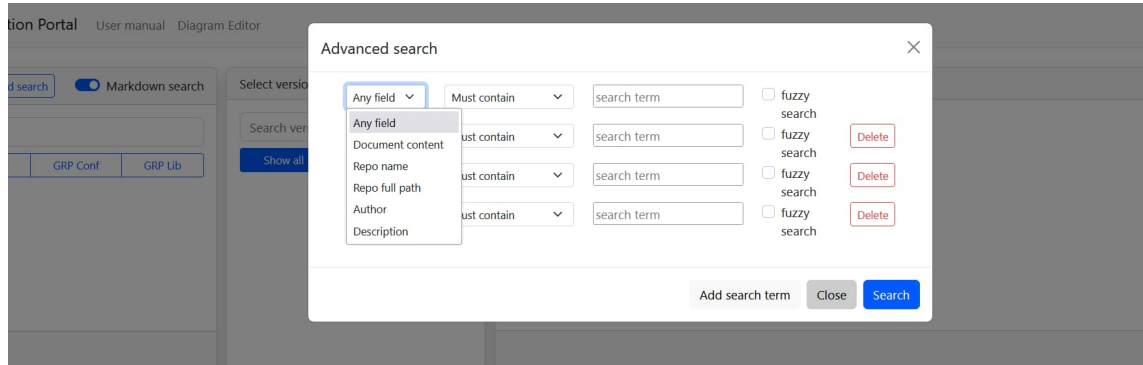


Figure 10: Advanced search UI

5.6 Tuning relevance scores

The search results from OpenSearch usually have a relevancy score attached to them. This allows ranking of the search results based on relevancy. Relevancy calculation can be finetuned by giving some fields of the indexed documents more weight than others. This can be done independently for every search, but in this project the field weights are set in SearchManager class and affect every full text search, that searches in every field, in the same way.

First attempt at weighting the fields was to order them from most to least relevant, based on intuition, and then give the least relevant field no boost, the second least relevant field boost of one, third least relevant boost of two and so on. The relevance tuning is very depended on the use case and cannot be easily translated to other domain. The order of relevance for the fields is show in table (3).

Table 3: First attempt at setting boost values for document fields

Field name	Boost
repository	4
author	3
description	2
tags	1
text_body	0

This arrangement had some faults when testing it. The repository names follow a naming convention where the place of the repository in the hierarchy of all Git repositories is expressed in the name of the repository, with every level separated by a forward slash symbol. The last part of the name is often unique and is what the user uses as a query when searching for the documentation of that particular repository, instead of using the whole hierarchy. Searching for the repository name like this works in the old search engine, since it only matches with the name of the repository, and not any other field. In the new search engine, with the initial relevance tuning, these kinds of searches often ranked repositories with the searched name lower than the ones that contain lots of mentions of the searched term in their other fields, like description and body text.

This was fixed by including an additional field for shorter repository name and boosting that greatly, which almost guaranteed that a repository would end up having the highest relevance score when its name, even without the hierarchy, is used as the search term. This new order is shown in table (4), below.

Table 4: New boost values with an additional field

Field name	Boost
repo_short	1000
repository	4
author	3
description	2
tags	1
text_body	0

6. EVALUATION OF RESULTS AND FUTURE DEVELOPMENT

A prototype application was successfully created using the described design and it was deployed on a QA (*Quality Assurance*) server which has the same kind of environment that the production server has. OpenSearch was deployed as separate microservice, with 3 nodes, using docker-compose to orchestrate the system.

Some tests were made there to see if the resulting design satisfies the requirements set in subchapter 4.1 and to answer the research question Q2. There is also evaluation of effects on security to answer security question Q3 on this chapter.

6.1 Effects on resource usage

The effect on resource usage was studied under light load on the QA server, when indexing was done for one user while another user was using the service. By looking at the statistic produced by Docker. While indexing, the memory usage of the application periodically rose from around 115 Mb to around 124 Mb. The OpenSearch nodes consumed total of 2,5 Gb of memory during the test and their core use mostly stayed around 4%, but occasionally jumped up to 20%-40% for less than a second.

Executing a search with the old search feature caused the CPU use of the application to only rise for around 1%, but with the new search feature one of the OpenSearch containers and the app both saw momentarily spike of 20 percentile units in their CPU use.

6.2 Effect on speed

Tests on the speed of the search feature were done by inserting code in the client side Javascript that measured the time it takes from user starting the search to use getting the results displayed in the client. Searches were then executed using the web UI as a normal user would. As the Gerrit API is the bottleneck when it comes to speed, and its response time varies for reasons that can't be affected by the application, the results are hard to replicate.

Old search engine returned 5 results with response times that averaged around 1200 ms and 1 result with 300 ms. When user selected a project from the results to display branches and tags inside it, the response took on average 1200 ms.

With the new search engine 5 results took 1900 ms and one results took on average 1000 ms. When user clicked a repository to display branches and tags inside it, this operation took on average 75 ms. This was because with the new search engine, the information about the tags and branches were already sent to the client when the search was first executed. After this initial operation, which caused the search to respond slower at first, was done, going through the results was faster.

With the combination of slower initial search and faster going through the results, it can be considered that requirement 4 is satisfied.

6.3 Estimated effect on security of the system

As the new search feature modifies the system heavily, effects on security should be considered. Requirement 2 states: “No data is leaked to 3rd party servers.” and requirement 3: “Users shouldn’t be able to access material that they don’t have permission to see”.

Since the application with the new search engine and the OpenSearch containers, runs in the same docker-compose network, it is easy to control what parts of the application are possible to access from the outside the network created by docker-compose. As a default, docker-compose doesn’t allow connections to the set of containers launched by it, called “swarm”, from outside the swarm. The containers running in the same network can connect to each other and they are by default in the same network. [24]

Custom network was also created in the docker-composes configuration file, so that the OpenSearch containers can be only accessed the containers that need to access it. On the production server, password authentication is taken into use, so that if something malicious manages to get running in the container swarm and the network of the OpenSearch containers, it would still need to know the password to be able to access the data in the OpenSearch containers.

6.4 Future development

When tested on QA server, the prototype was also subject to examination by other developers and other team members. There are already some ideas for future development:

- Doxygen support: As of writing this, support for Doxygen documentation was just released for Documentation Portal. Doxygen is technology which renders HTML documentation from source code and Documentation Portal is able to

integrate that documentation with the possible Markdown documentation seamlessly with its build pipeline. The new search feature might be later extended to search for Doxygen documentation too.

- Improved UI. Now the Advanced search window only allows for a small subset of Lucene query syntax to be used. This could be extended in the future.
- Performance improvements on indexing and searching. Experimenting with cached search results using, for example, Redis cache.
- If the “tags”-feature of Documentation Portal gets implemented and popular, support for that is fast to implement into the search feature, since one field in the index is reserved for that information.

7. CONCLUSION

The problem of upgrading the search engine for the Documentation Portal application was approached following the design research method. Problems with the old search feature were noted and ideas on how to improve it with a new solution were refined into the form of requirements. Based on the requirements for the new search engine and the current architecture of the Documentation Portal application, OpenSearch was chosen as the search engine component that the new search feature would be designed around.

A design was created for integrating the new search engine into Documentation Portal application. It was implemented and tested on QA server to see that it works and satisfies the requirements.

First research question is: “What are the requirements for the implementation of the search engine component and what kind of architecture should be designed based on those requirements.”. Set of requirements for the solution was defined in subchapter 4.1. They were ordered by importance and a search engine component was selected, that was considered to be able to function as a base for a design that can fulfil the requirements. The architecture and function of the implemented design is detailed in chapter 5.

Second research question is: “How to limit the access of users to only the documentation that they are allowed to see.”. This was done with a method that sacrificed some performance for added safety: Every search result from the OpenSearch index is checked against a search result from the Gerrit version control system in the backend side. This way the user cannot see any search results that the user doesn't have access to in the version control system.

Third research question is: “How is resource usage of the system affected by the change?”. This was discussed in subchapter 6.1 and the resource usage was not greatly increased with the number of users and documentation that was used to test the application on the QA server. However, the resource usage should be monitored closely when the feature is deployed into wider use and actions should be taken if it increases too much.

The first version of the new search feature satisfies the requirements, but there are still many ideas for future development and it hasn't been tested with large number of actual users.

8. REFERENCES

- [1] A A G Y Paramartha, L J E Dewi, The Development of search engine service for official academic documents, IConVET 2020 Journal of Physics: Conference Series
- [2] Apache Lucene website, <https://lucene.apache.org/core/>, accessed 28.06.23
- [3] Amazon AWS website, "What is OpenSearch", <https://aws.amazon.com/what-is/opensearch/>, accessed 28.06.23
- [4] Brave.com, Search engine Definition and meaning, <https://brave.com/glossary/search-engine/>, accessed 27.08.23
- [5] N. Deepa, B. Prabadevi, L.B Krithika, B. Deepa, An analysis on Version Control Systems, 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)
- [6] DB-Engines.com, DB-Engines Ranking of Search Engines, <https://db-engines.com/en/ranking/search+engine>, accessed 27.08.23
- [7] Design Research in Information Systems, A. Hevner, S. Chatterjee, Springer, 2010, ISBN: 978-1-4419-5652-1
- [8] Dictionary.com, Search engine Definition and meaning, <https://www.dictionary.com/browse/search%20engine>, accessed 27.08.23
- [9] Docker web page. <https://www.docker.com/resources/what-container/>, accessed 27.02.23
- [10] Encyclopedia Britannica, Markup language definition, <https://www.britannica.com/technology/markup-language> , accessed 30.10.23
- [11] Essentials of Microservices Architecture: Paradigms, Applications, and Techniques, S. Chellammal, G. Gopinath, P. Raj, 2019, ISBN: 9781000617566
- [12] Flax.co.uk, ElasticSearch vs. Solr performance: round 2, <https://www.flax.co.uk/index.html?p=2826.html>, accessed 27.08.23
- [13] Gerrit code review web page, <https://www.gerritcodereview.com/about.html>, accessed 02.03.23

- [14] Gerrit – Concepts and Workflows, presentation, <https://docs.google.com/presentation/d/1C73UgQdzZDw0gzpaEqIC6SPujZJhqamyqO1XOHjH-uk/edit?usp=sharing>, accessed 03.04.23
- [15] Git book, 2nd Edition, online version, Apress, S. Chacon, B.J Straub, 2014, available: <https://git-scm.com/book/en/v2> , accessed 10.04.23
- [16] Github, Terrier license, <https://github.com/terrier-org/terrier-core/blob/5.x/LI-CENSE.txt>, accessed 27.08.23
- [17] IBM, What are containers?, <https://www.ibm.com/topics/containers>, accessed: 27.10.23
- [18] Kroki web page. <https://kroki.io/>, accessed 27.02.23
- [19] Linuxlinks.com, 8 Best Free and Open Source Search Engines for Big Data, <https://www.linuxlinks.com/searchengines/>, accessed 28.08.23
- [20] L.R.E Quin A Comparison of the strengths of some widely-used markup systems, Balisage: The Markup Conference 2014
- [21] Mabl.com, Choosing between Elasticsearch and Solr, <https://www.mabl.com/blog/choosing-between-elasticsearch-and-solr>, accessed 27.08.23
- [22] M. Fowler, J. Lewis, Microservices a definition of this new architectural term, <http://martinfowler.com/articles/microservices.html> , accessed: 28.10.23
- [23] MkDocs web page. <https://www.mkdocs.org/>, accessed 27.02.23
- [24] Networking in Compose, <https://docs.docker.com/compose/networking/>, accessed 06.11.23
- [25] Documentation Portal user manual
- [26] OpenSearch documentation, <https://opensearch.org/docs/latest>, accessed 28.06.23
- [27] P. Di Francseco, P. Lago, I. Malavolta, Migrating Towards Microservice Architectures: An Industrial Survey, 2018 IEEE International Conference on Software Architecture (ICSA) [MICRO1]
- [28] Relevant search : with applications for Solr and Elasticsearch, Turnbull, Doug, author. Berryman, John, 1980- author.; Grainger, Trey, writer of foreword, 2016, ISBN: 1-63835-361-1, [RS]
- [29] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, A. Ivanov, System-on-Chip: Reuse and Integration, Proceedings of the IEEE, Vol. 94, Issue 6.

- [30] S. Robertson, H. Zaragoza The Probabilistic Relevance Framework: BM25 and Beyond, Information Retrieval Vol. 3 No. 4 2009 333-389
- [31] Sphinx documentation, <http://sphinxsearch.com/>, accessed 21.09.23
- [32] S. Xiujin, W. Zhenfeng, An optimized full-text retrieval system based on lucene in oracle database, Proceedings - 2nd International Conference on Enterprise Systems, ES 2014, 2014, p.61-65
- [33] Terrier documentation, <http://terrier.org/>, accessed 21.09.23
- [34] T. Mikkonen, C. Pautasso, K. Systä, A. Taivalsaari, Cargo-Cult Containerization: A Critical View of Containers in Modern Software Development, 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)
- [35] Understanding Information Retrieval Systems: Management, Types, and Standards, Auerbach Publishers, Incorporated, Marcia. J. Bates, 2011
- [36] Xapian, Licensing, <https://trac.xapian.org/wiki/Licensing>, accessed 27.08.23
- [37] Xapian, Website <https://xapian.org/>, accessed 05.12.23
- [38] Z. Lu, C. Chen, J. Xin, Z. Yu, On the Auto-Tuning of Elastic-search based on Machine Learning, Conference on Control, Robotics and Intelligent System (CCRIS 2020), 27.10.23