Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

VINCENT FEDERLE
MASTER THESIS

Searching within EDITIVE

Evaluation and Implementation of a Reference Architecture

Submitted on 18.05.2021

Supervisor: Dipl.-Inf. Hannes Dohrn, Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 18.05.2021

License

_____

Erlangen, 18.05.2021

Abstract

EDITIVE is a platform based upon the collaboration principles first spread by GitHub. In such a multi-level content collaboration platform, a search functionality is useful and often demanded. The main reasons therefore are to reduce complexity and provide advanced information retrieval, including content meta-information. The difficulty to provide a search functionality within such a platform relates to the underlying content data model and its complexity. Atomic data structures enable more search precision and contribute towards an effective search implementation. EDITIVE runs on a mainly atomically defined data structure. The highly complex data model within EDITIVE stems from the flexibility based on the Git collaboration principles offered to the user, merged with advantages of wikis.

This master thesis shows the difficulties of implementing a search functionality within a multi-level content collaboration platform, more specifically EDITIVE. It presents the design and implementation of it in the EDITIVE context based on the search engine Apache Solr. This thesis shows a reference architecture implemented on Solr. Furthermore, it elaborates on the utilization of the underlying data model. We advise several ways on how to further refine and extend the resulting search implementation beyond the scope of this thesis.

Contents

1 Introduction

Expectations towards software applications have grown in the direction of easy usability, shallow learning curves and intuitive designs (Gupta et al., 2017). Nevertheless, professional tools are expected to provide a high set of features and flexibility depending on their appliance. The necessity of information retrieval has grown with the increasing amount of data available (Roshdi et al., 2015). Additionally, problems such as data security and safety are growing more important because sensitive information is increasingly stored in a digital format. Protected information is stored within search indexes and must be secured accordingly. On top of that, software code should facilitate proper maintenance to ensure a long lifetime (Abdullah et al., 2017).

EDITIVE enables new ways of collaboration (EDITIVE, n.d.) and therefore new layers of complexity for information retrieval considering security, usability and performance. This thesis showcases a reference architecture and its evaluation and implementation based on Apache Solr (Apache, n.d.), which we will further refer to as Solr. The resulting artifact follows state-of-the-art coding guidelines. It provides a basis for high usability and performance, which can be further configured and extended. Some possible extensions are described in the outlook section of this thesis. The result of this thesis consists of:

- An architecture and implementation of a search component fulfilling requirements towards an EDITIVE search functionality.
- Validation of requirements based on an evaluation schema.
- An outlook on the search functionality development beyond the scope of this thesis.

EDITIVE is in some ways similar to but also unique from other platforms. There is no across-the-board implementation that can be utilized, but many search platforms provide a large spectrum of functionality out of the box, which this thesis also utilizes. EDITIVE's business model is mainly structured around flowing text compared to a source code management platform such as GitHub, which might also change the expectations towards its search functionality. In this work, we provide a possible implementation and architecture for a search component integrated within EDITIVE.

The thesis is structured as follows: Section 2 focuses on requirements towards a search feature embedded in a multi-level content collaboration platform. It presents an evaluation scheme for its requirements. In section 3, current technologies are discussed and the reasoning for choosing Solr is explained. Additionally, a brief overview of the GitHub search (GitHub, n.d.), the Wiki Object Model (Dohrn et al., 2011), short WOM, and difficulties for a search implementation are outlined. Section 4 leads through the design and reference architecture for a search implementation. Section 5 briefly concludes on the implementation, continuing with the evaluation of the architecture and implementation in section 6. Subsequently, an outlook is outlined in section 7. Section 8 concludes the scope of this work.

2 Requirements

The availability of applications improved over the years (Robillard et al., 2009). Companies such as Microsoft decided to focus more on open-sourcing applications including the well known Windows calculator among many others. With more and more tools contributing to a better development process, complex solutions can be realized. But whether those solutions provide value to users often depends on solid requirements. Therefore, we need to gather and formulate requirements to be able to build an effective solution. Requirements mostly dictate the direction an implementation takes (More et al., 2011). This section is structured into the purpose of search functionality, functional requirements, non-functional requirements and an evaluation scheme to validate these requirements.

2.1 Purpose of search in a multi-level content collaboration platform

Platforms such as YouTube, Facebook, Netflix and Reddit are successfully employing recommendation and subscription systems. The fast consumption of entertainment and information nowadays requires such systems and these systems become relevant even within the field of software engineering (Robillard et al., 2009). These recommendation and subscription systems don't replace searches, in fact users expect the existence of search functionality within most applications. The trend shows, that search functionality became simplified over the last years. There are fewer search options, facets, filters and topics, even up to the point of offering a search bar without any further possibility of configuration. Additionally, users increasingly rely on high precision in the top few search hits, partially caused by the growing impatience of users (Lown et al., 2013). As Steve Lohr describes in 2014, even an "Eye Blink" is too long for a user to wait while querying e. g. Google (Lohr, 2012).

An effective search implementation seems to be difficult to achieve under these assumptions, but indexing and search libraries such as Lucene enable effective implementations given these requirements. Therefore, even smaller products can provide search functionality with a manageable amount of effort.

Search functionalities can provide many benefits to a multi-level content collaboration platform. First, users will find relevant content faster and can also discover new content of other collaborators, maybe even exploring content. Second, search metrics can be documented and used for further product development. Third, mobile users don't like complex navigation, since they have a smaller device than desktop users to operate on. A search functionality simplifies navigation. Lastly, users nowadays simply expect a search functionality within most applications (Aliannejadi et al., 2018). These benefits, especially the first, justify the implementation of a search functionality within EDITIVE.

2.2 Requirements for the search component

The requirements listed in this thesis are mostly based on internal company design. Some requirements are influenced by potential customers of EDITIVE as relayed by the company staff. The small startup EDITIVE, based in Erlangen – Germany, doesn't have a large user base at this point in time but is currently focused on gaining momentum in the market. Most functional requirements have been defined on assumptions of user behavior, user expectations and other comparable existing search features offered by popular products. Non-functional requirements have been aligned with the company's technical specialist.

EDITIVE's data model is comparable to a Git based data model. It consists of commits containing changes on different branches, therefore enabling versioning in addition to high col-

laboration. Tags can be assigned to specific versions, acting as snapshots of a branch in a specific version. The main elements of EDITIVE's contents are Eddys and their documents. Eddys represent projects which can contain many documents. These projects can be forked to a separate representation and can be modified on multiple branches and versioned with multiple tags. Any commit represents a version of none or multiple documents and can be restored at any point in time.

2.2.1 Functional requirements

The goal is to be able to search through most of the content within the *HEAD* commits of all projects in the version representing the most recent content. The search should be performed with the following information. The user can retrieve this information from the search results:

User editable information:

- The descriptions and names of Eddys.
- Eddy documents' names and contents.
- Authors, creating and modifying content within EDITIVE.
- Branches within Eddys.

User non-editable information:

- Names of commits represented by their commit-hash.

When retrieving an Eddy, all Eddy related information is accessible from the result of the search. From there on, the user can navigate to the Eddy and will be able to further navigate from there to specific branches or versions. Documents retrieved from searches are in the *HEAD* version for each Eddy and fork the documents are referenced in.

Additionally, issued searches must respect the visibility settings of projects and consider the searching user's permissions to ensure no information is leaked in any unauthorized way. Regarding usability, the users should be able to decide whether they want to search solely through Eddy information, Eddy document information, or both. Documents are indexed completely to reduce complexity. They will be stored in a plain text form instead of their WOM extensible markup language (XML) representation.

2.2.2 Non-functional requirements

The search implementation should be done following the "kiss"-principle. The design of the search functionality should follow a simplistic design and can be initially implemented without any third party software. It doesn't require to be scalable in its first iteration. From this starting point on, the search functionality should be refined and improved incrementally. Non-functional requirements towards the design and implementation are:

- The software artifact needs to be easily maintainable and extendable. For that, all coding functionality needs to be unit tested. Integration tests are expected where of use. Additionally, the code should follow best practices, fulfill EDITIVE Checkstyle rules and follow internal coding guidelines.
- All programming code needs to be written in Java. Configuration files should be in a common format, e. g. XML. These files are restricted to the backend service and should be running within docker-compose for local setup and Kubernetes for cloud setup. The resulting objects must be easily processable by the frontend. The frontend implementation is not the scope of the search implementation.
- Written Java code follows the principles of object-oriented design.

- It is recommended to use as many existing dependencies as possible introducing new dependencies only when required for core functionality. Emphasis is placed on implementing specific functionality on the existing dependencies instead of introducing new dependencies to the project. This prevents unnecessary governance and maintenance of marginal dependencies as well as reduces project complexity.
- All the search component functionalities such as indexing, searching and starting of services can be toggled on or off by a switch. The EDITIVE backend must remain fully functional if the search is disabled. In this case, search queries should return an empty result set. This also benefits towards a separation of concerns and resilient software design by enabling possible implementations to cope with ways of failure.
- Program code should allow for easy profiling, that can be toggled on or off. This helps to validate implementations and therefore improves further extension and testing of the functionality.
- The implementation should follow a synchronous approach for indexing and searching in order to reduce complexity by possibly compromising on response time.
- All relevant Eddys and documents must be retrievable by the search implementation. This implementation does not focus on precision and recall, but instead focuses on indexing and reading performance. Precision and recall can be analyzed effectively by utilizing metrics and query logs, thereby requiring a user base (Zhou et al., 2017). Existing search engine frameworks can be configured and fine tuned to deliver a relevant ranking of hits.
- A modular structure of the functionality enables simple replaceability. The implementation utilizing a search platform can be switched out with low effort and without change to the non technology related implementation. This requirement might change since EDITIVE is starting to gain traction in the market.

2.3 Evaluation scheme for requirements

Based on the requirements, we define some criteria of quality for the search component. These criteria will be used to validate and evaluate the requirements. The criteria are chosen from the ISO/IEC 25010, that replaced the previously established standard ISO/IEC 9126 (ISO, n.d.).

- The **functional suitability** of the implementation is measured by the set of functions covering the functional requirements listed in 2.2.1. Additionally, the implementation provides correct results. Lastly, the implementation facilitates the accomplishment of the user's tasks.

- The time behavior is measured in order to evaluate **performance efficiency**. Indexing and searching both fulfill the specified performance requirements and also scale in a reasonable way under the assumption of higher capacity.

- **Compatibility** is measured by the co-existence and interoperability with the existing EDITIVE implementation. Passing integration tests on the EDITIVE backend, including the search components as well as successful deployment builds, indicate the integration and compatibility.

- An appropriate user error protection is established. The interface is simple to use and easy to understand. Additionally, the search implementation is transparent to the user thereby encapsulating internal complexity. The search component has a high degree of **usability**.

- **Security** measures are implemented in order to prevent the user from manipulating data in an unauthorized or unintended way. Also, users only retrieve information compliant with Eddy visibility and access grants.

- Good **maintainability** of the search component code is required, especially regarding the modifiability. Establishing a high degree of maintainability is required because the recommended architecture and implementation will represent a first iteration of the search functionality. It should be capable of modification and configuration to fit future requirements. Unit tests are provided for all implemented logic to also imply modularity. Simple profiling is implemented in order to make the software analyzable and to make future modifications comparable to current implementation.

- In this context we also place high value on the **portability**. There are many other technologies this architecture can be implemented on, but the architecture might change slightly due to the choice of technology. The many search platforms available are specialized to cover a wide field of application areas and therefore an appropriate framework is carefully chosen in regards to the requirements. The design separates the implementation of the specific search engine framework from the implementation required in the EDITIVE context. This should improve the replaceability of the search engine framework implementation and technology by keeping the EDITIVE specific non-framework related implementation.

3 Technology and state of the art

In this chapter, we will further examine Apache Solr and why it was chosen for the EDITIVE search implementation. Choosing Solr also has an impact on the architecture and implementation of our solution, therefore chapter 3.1 is further separated into a description of the Solr architecture, our reasoning for choosing Solr and an overview of few optimizations and best practices using Solr. In chapter 3.2 we briefly look at the GitHub search, which is implemented using Elasticsearch. Then in chapter 3.3, the WOM is briefly presented as the basis of EDITIVE documents' internal storage. Finally, chapter 3.4 presents some difficulties in designing the search functionality.

3.1 Apache Solr

Apache Solr is an open-source search platform with an active development community and regular releases. It is well established for enterprise search and designed for its scalability and fault tolerance, utilizing index replication and distributed search (Apache, n.d.). Solr is based on the open-source search engine library Apache Lucene. Lucene and Solr merged in 2010 and the same committers are further developing Lucene as well as Solr. Both, Lucene and Solr, are written in Java. Some of Solr's major features relevant for this thesis are its "Advanced Full-Text Search Capabilities", "Near Real-Time Indexing", "Flexible and Adaptable with easy configuration", "Highly Scalable and Fault Tolerant" and "Easy Monitoring and Optimized for High Volume Traffic" (Apache, n.d.).

There are currently two modes to run Solr. First is the SolrCloud mode, which offers many features which are automatically coordinated by Apache ZooKeeper and Solr. These include functionalities such as index replication, load balancing and distributed queries relevant for fault tolerance and performance. Second is the standalone mode. The standalone mode also offers these possibilities similar to the SolrCloud mode, but they have to be implemented and handled manually by using third party tools or self-made implementations. For some use cases, the standalone mode can be preferred, but these don't include any requirements currently set for the EDITIVE search. Additionally, the SolrCloud mode can always be reduced to standalone, if it is too heavyweight. The flexibility and improved utility it provides are desirable in a startup, which isn't that established on the market yet. Concluding, we will only use the SolrCloud mode for our reference architecture and implementation.

The chart below shows a simplified view of the SolrCloud architecture. SolrCloud nodes are all independent from each other, they don't use a master-slave architecture compared to the standalone mode. Every node can use ZooKeeper to learn about the state of the cluster. One Solr node can host multiple cores and within each of these cores runs its own Lucene engine. SolrCloud uses collections to define the data schema. Collections are logical unions of shards and each shard is represented as a core inside of a node. The replication factor of Solr and the number of shards determine the number of cores used. For example, with a configured replication factor of 2 and a collection distributed across 3 shards, we will then require a total of 6 cores (replication factor 2×3 shards). All data is stored on the local file system. Solr Cloud also provides many metrics and monitoring capabilities. These have to be explicitly activated by starting a Solr metrics process which is a separate process within a Solr node.

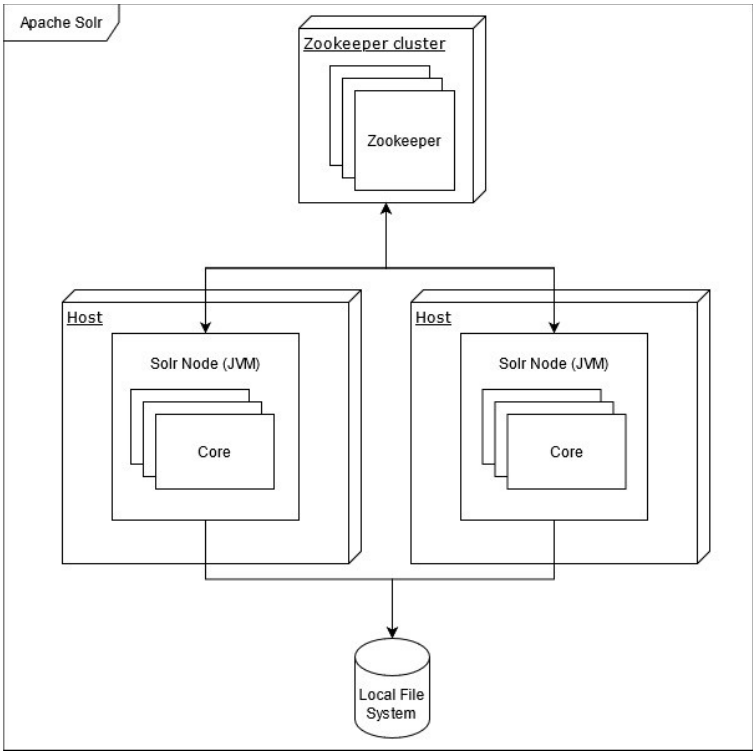

Figure 3.1: Apache Solr SolrCloud architecture

The decision towards Apache Solr is based on multiple criteria. Due to our requirements, the focus on technology is reduced, since the technology should be easily interchangeable. Most of the code is required to be portable to a different concrete implementation and search platform.

- Licensing of the software.
- The degree of establishment within enterprise applications.
- Possibility for support & present up to date documentation.
- Active product development with frequent and recent updates.
- Needs to be capable to fulfill all requirements towards our search component.
- Java support.

There are many established search platforms on the market, but Solr, being one of the most established platforms, was chosen based on the criteria mentioned above. Elasticsearch was also a highly considered choice. Both Elasticsearch and Solr are utilizing Lucene and are written in Java. Due to the previous work on a complex search implementation within EDITIVE (earlier Sweble) by Robert Miller (Miller, 2018), which was implemented utilizing Elasticsearch, we decided to choose Solr and its simple Java interface SolrJ.

Solr enables us to fulfill all requirements in an efficient way. It has high potential and offers the capability to be configured and optimized for most use cases. Utilizing its metrics to further evaluate configurations can ease the validation and implementation. At the core of Solr's performance is its **schema design**. Here we carefully need to decide on appropriate **field definitions** to enable dynamic flexibility or a more optimized stricter schema. Furthermore, an effective schema design enables Solr to make best use of its text analysis features (Apache, n.d.). It is capable of tokenization, stemming, handling synonyms, removing stop words, even sound-like analysis and partial indexing. Spell checking can be configured for a domain specific language checking. Solr's XML support could be of interest because one of the native and descriptive formats of Eddy documents is XML, but more about that in section 3.3.

Enhanced searching also provides benefits to performance, precision and recall. Solr provides a lot of functionality and built-in functions, which can be utilized in queries. Additionally, it allows to set up result rules, query and function boosting. Solr is well known for its **faceting** functionality to further enhance navigation.

3.2 A look at the GitHub Search

The GitHub Search application programming interface (API) is implemented utilizing Elasticsearch. It offers several representational state transfer (REST) APIs as well as a simple graphical user interface (GUI). It makes use of Elasticsearch's ranking, ordering result hits by their relevance. Additionally, it enforces rate limiting and other limitations as query length and operator count. Queries can run into timeouts and will then only return the hits that were identified until its timeout (GitHub, n.d.).

The GitHub Search API differentiates between searching code, commits, issues & pull requests, labels, repositories, topics and users. It has separate REST methods for each of these search cases. The search bar in the GUI queries all of these interfaces and provides results ranked by their relevance score within their respective categories. Searching in the code category requires to provide a login. Facets can be further utilized within the GUI to navigate to results in specific programming languages.

The current GitHub Search implementation also meets critique regarding simplicity. An advanced search can provide many options but some users express their confusion or criticize requiring too many steps to reach their goal. This is further described in an issue thread on GitHub (GitHub, 2017).

3.3 Wiki Object Model (WOM)

Eddy documents are internally stored in the extensible wikitext markup language (XWML) and can be read and modified using the WOM. These documents contain different parts that can be further distinguished. See a simple example below containing a body with a paragraph and some text.

```
 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<article xmlns="https://schema.editive.org/editive-article-data-model/
1.0.0" version="1.0.0">
    <body>
       <p xmlns="https://schema.editive.org/editive-hypertext-data-model/1.0.0">
          <text xmlns="https://schema.editive.org/editive-resource-data-model/1.0.0"
                xmlns:ns1="https://schema.editive.org/editive-article-data-model/1.0.0">
            Richtlinien zum Vollzug der Zweiten Bayerischen Infektionsschutzmaßnahmenvero
rdnung an den bayerischen Universitäten</text>
        </p>
    </body>
</article>
```

Snippet 3.1: WOM document

Regarding a search implementation, this document structure enables further faceting and more precise searching by utilizing the different defined sections of a document. This would increase the complexity of the search implementation, but could also provide more features and

flexibility. By reducing a WOM document to its text representation, the above document is reduced to the string:

```
Richtlinien zum Vollzug der Zweiten Bayerischen Infektionsschutzmaßnahmenverordnung
an den bayerischen Universitäten
```

Snippet 3.2: WOM document content text

The current goal of the EDITIVE search is not a navigation from paragraph to paragraph, but a navigation towards Eddys and their documents.

3.4 Difficulties in designing search functionality

Taking a look at the Google search functionality again the user is only confronted with a single search bar. After issuing the first search, Google retrieves all hits within less than a second and offers further navigation, utilizing its faceting functionality. These facets include a few categories such as “News”, “Images”, “Videos” and “Shopping”, offering a few filters regarding languages and time. With these few features, Google is dominating the search market, even having an active lawsuit initiated by the U.S. Justice Department. The lawsuit is charging that Google holds an illegal monopoly over search by controlling 88 percent of general searches and therefore stifling competition (Office of Public Affairs, 2020). So even with such simple functionality offered to the end user, Google has such a dominant position due to its accurate and precise algorithms combined with high performance. Due to this dominant position, it has shaped user expectations even towards libraries (Lown et al., 2013). One might argue, that this has no relevance for EDITIVE, since the requirements for Google’s search and EDITIVE’s search are not comparable. The complexity of Google’s search is much higher and its amount of content exceeds EDITIVE’s by an immense magnitude. But we can still derive some of the users’ current expectations towards a search system from it. By keeping the search interface simple and focusing on high precision and recall instead, the EDITIVE search can be more useful for the user and improve the overall value of the product. This can be accomplished in a limited way in the scope of a master thesis, but the basis for this can be created, which can then be further improved upon by analyzing user behavior.

4 Architecture and design

This chapter describes the design for the search functionality in EDITIVE. Chapter 4.1 contains information on the EDITIVE systems architecture as well as a proposed architecture for embedding search functionality within EDITIVE. To define a systems architecture, design patterns and architectures are analyzed. Furthermore, we describe potential modules within an architecture for the EDITIVE search functionality. Next, a sequence flow is discussed to concretize the behavior. Chapter 4.2 describes specifics regarding the search functionality and its design and architecture, based on Solr using the embedded architecture from chapter 4.1. Additionally, chapter 4.2 focuses on a target architecture for Solr within EDITIVE. It only describes and utilizes best practices and currently required functionalities.

4.1 Systems architecture

There are currently many architectural patterns as well as several ways to describe a designed architecture. The work from Niu et al. (Niu et al. 2013) evaluates software architectures and introduces methods to determine appropriate architectures, which will be utilized in the following chapters. We elaborate the design based on a few standardized diagrams using unified modeling language (UML) and other established methods. Additionally, we decided to keep the architecture description focused on the impact of non-functional requirements. Software architecture includes conflicting objectives and, at a system level, all of these have to be considered. Because of these conflicting objectives, we can only design and evaluate a system architecture based on its requirements (Niu et al. 2013). The conflicting objectives include scalability vs. reliability. Methods such as backup strategies and data replication can be implemented to fulfill reliability, but at the same time they will reduce the ability to scale the system. This scalability is reduced because all replication and backup strategies have to be scaled as well. This impacts the resulting cost of operation or, within a given set of hardware, both of the two objectives require more resources without improving the adversary objective.

4.1.1 Architecture patterns

Patterns in the domain of software engineering are general and reusable solutions to commonly occurring problems. Architecture patterns in specific have a broader scope compared to software design patterns.

Since synchronous search implementations can slow down the application’s performance, it is a common practice to provide asynchronous indexing and sometimes even asynchronous searching. Updating an Eddy document leads to saving new commit data to the database (DB) and likely also leads to an indexing of said data into the search index. If this is done synchronously and the time consumption by the DB access is noticeable, then the indexing time of the search will noticeably add on top. Therefore, an event driven architecture can provide good performance. Additionally, it provides high flexibility and loose coupling of different event-consuming processes. Below, an event architecture is displayed containing some of the main search functionality processes.

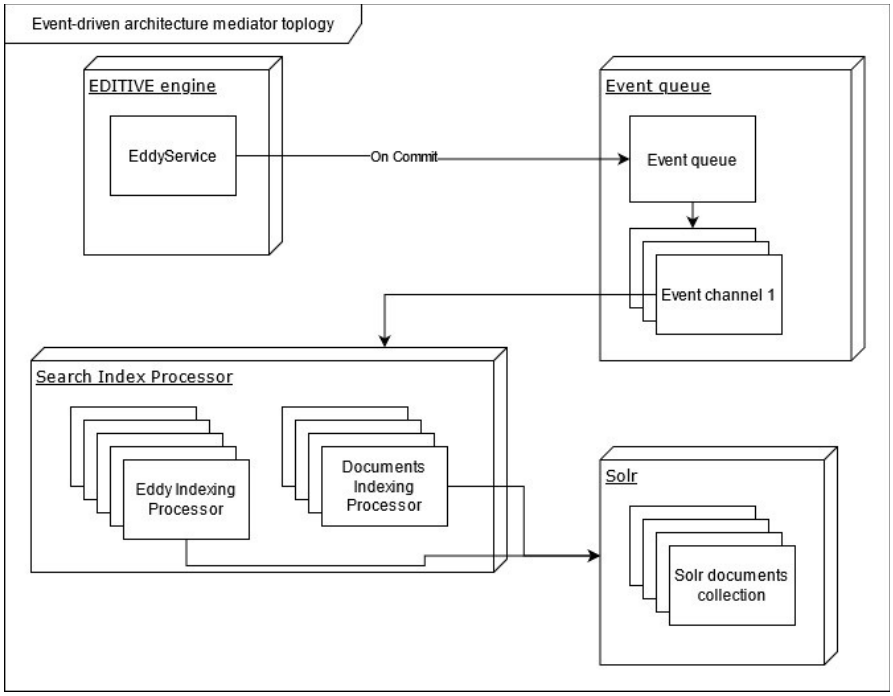

Figure 4.1: Event-driven architecture mediator topology

This event-driven architecture utilizes the mediator topology, providing event channels towards specific processors. There would be an event channel for Eddy related events and an event channel for Eddy documents related events. This could be extended to also include an event channel for events such as indexing tags within EDITIVE. This is accomplished by utilizing a broker topology. The difference is the absence of an event mediator redirecting events from the event queue directly to the corresponding event channels. Instead, events from the queue are consumed by any processor and if the processor can't consume the event, it is forwarded to another specified event channel.

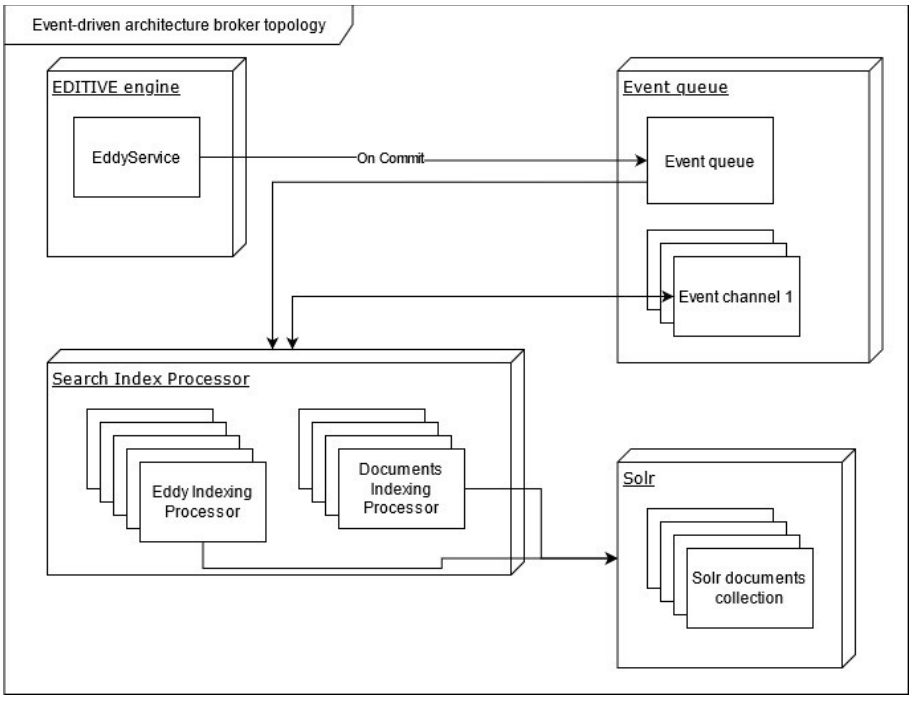

Figure 4.2: Event-driven architecture broker topology

An event-based architecture has many benefits for EDITIVE. It provides overall agility due to its loose coupling. It can be scaled by adding further processor instances or different processors. All processors can be separately run on suitable infrastructure if necessary, which could benefit performance. The downside is that testing within such an architecture is very difficult. The processors' functionality itself could be unit tested. But tests for other components or tests on a higher level in the test pyramid are difficult to implement and maintain. Development is more complicated due to asynchronous operations and higher complexity of understanding and perceiving the overall system. Currently, a notification is not required when the search request is completed. Most likely, the user will issue a search and then wait for its completion. The search functionality in EDITIVE is required to be fast, but doesn't need to scale to an extreme, therefore we see no need for an event-based searching architecture introducing further complexity.

In order to cope with this complexity and development problems, a layered architecture can be implemented. A layered architecture provides high testability, easy development and reduced complexity. Since these are very important for EDITIVE and required for the search functionality, we accept its downsides of tight coupling and dependent deployment. The downside of low scalability only affects the adapter we design integrated within EDITIVE, since Solr has its own ways of scaling which are not directly affected by our adapter architecture. The EDITIVE backend architecture can be identified as a layered architecture including service oriented architecture patterns. Therefore, it would reduce the complexity by not introducing a new architectural pattern into a module of EDITIVE.

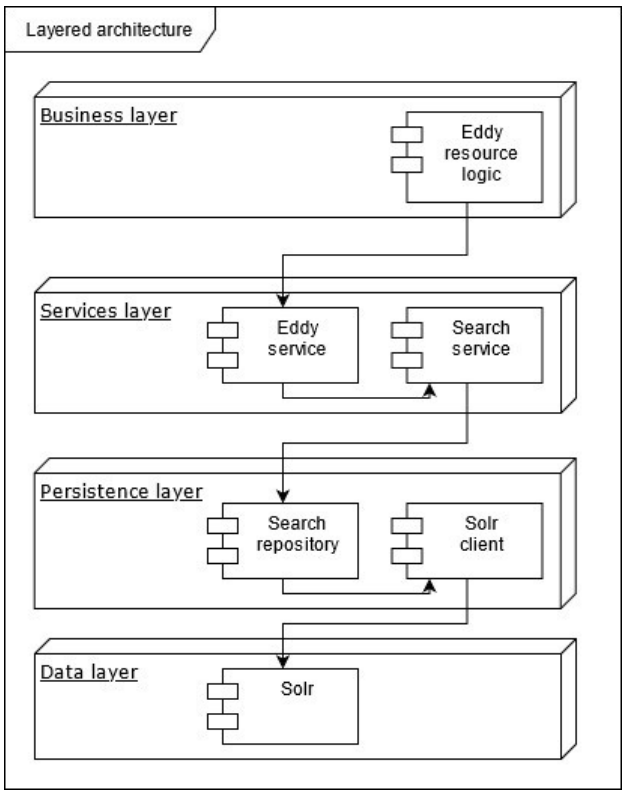

Figure 4.3: Layered architecture within EDITIVE

This general purpose architecture provides a starting point for EDITIVE's search functionality. It contains the adapter to Solr represented by a *search service* and a *search repository*. The *search service* functions as the interpretation of user actions within EDITIVE such as commits and it extracts relevant information for indexing. It also provides a simple interface towards other EDITIVE components issuing searches. The extracted information is then passed

on to a *search repository*, which is responsible for managing the *Solr client*. This architectural pattern supports the realization of the requirements towards an EDITIVE search functionality.

A microkernel architecture improves some issues implied by a layered architecture. It can be a useful pattern for incremental development by providing better deployment and loose coupling compared to the layered architecture. The microkernel architecture can be implemented within other architectural patterns. This is not within the scope of this thesis, but we recommend a microkernel architecture for further development of the *search repository* as shown in Figure 4.4:

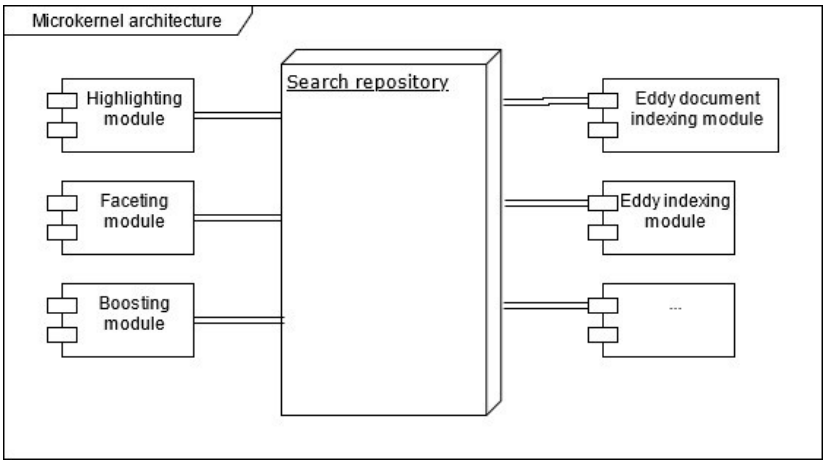

Figure 4.4: Microkernel architecture for the search repository

The central *search repository* represents the core system of the microkernel architecture enabling plug-in modules to be defined. For an architecture utilizing Solr within EDITIVE these plug-in modules can include a highlighting module, a faceting module or an indexing module. These modules are solely responsible for their task, enhancing overall agility, loose coupling and development of new features.

4.1.2 Search component embedded within EDITIVE

The search component embedded within EDITIVE is, on a high level, only concerned with two use cases:

1. A contributor changes content related data within EDITIVE.
2. A user issues a search within EDITIVE.

Other use cases, utilized by actors such as developers, include monitoring, testing and profiling but are not included as the core functionality. Nevertheless, these use cases are essential and the core functionality needs to enable or provide room for extension.

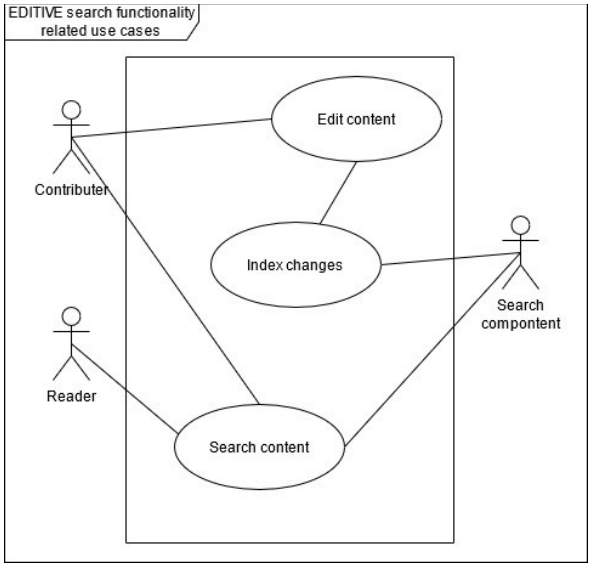

Figure 4.5: Use case diagram for the main use cases relevant for a search functionality

These use cases can be further detailed into editing Eddy document content or the direct editing of Eddys and the search for Eddys or Eddy documents. This can be further extended with the searching of authors, editing and searching of tags and commits.

A brief search architecture within the EDITVE system is designed. The EDITIVE is separated into multiple layers. It consists of the *frontend* (presentation layer), the *editive-service* (business layer), the *editive-engine* (service/business layer) and the *storage* modules (persistence layer). As mentioned in chapter 4.1.1, the search functionality should be located within the services and persistence layer.

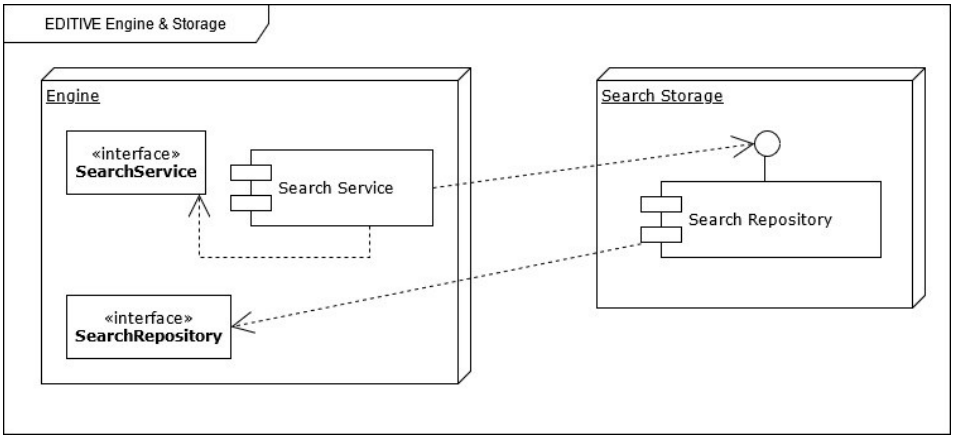

Figure 4.6: Java interfaces within EDITIVE's *Engine* and *Storage* modules

The approach displayed in Figure 4.6 is common practice within the EDITIVE implementation. The *Search Service* has no knowledge of the implementation and technology used within the *Search Storage*. Additionally, the *Engine* provides the interface that is implemented by the *Search Repository*. The *Search Service* operates on the interface, receiving the *Search Repository* implementation instance trough dependency injection. This prevents the *Engine* from depending on the *Search Storage*. By utilizing this method, further components similar to the *Search Storage* can be implemented. The current *Search Storage* would be an implementation on top of Solr, but future implementation could utilize other technologies and be easily toggled and compared. Since the *Engine* defines the interface it consumes, the expectations towards the *Search Storage* are clearly defined. The separation of these concerns simplifies testing, as well as enabling mocking of the used interfaces.

4.1.3 Indexing and searching sequence within EDITIVE

With a sequence flow, we can further concretize the two core use cases of indexing and searching. First, we describe the case of synchronous indexing in the EDITIVE environment. Indexing occurs during any content change, but the most complex changes to Eddy documents occur within commits. Therefore, we will mostly focus on commits within EDITIVE concerning the indexing.

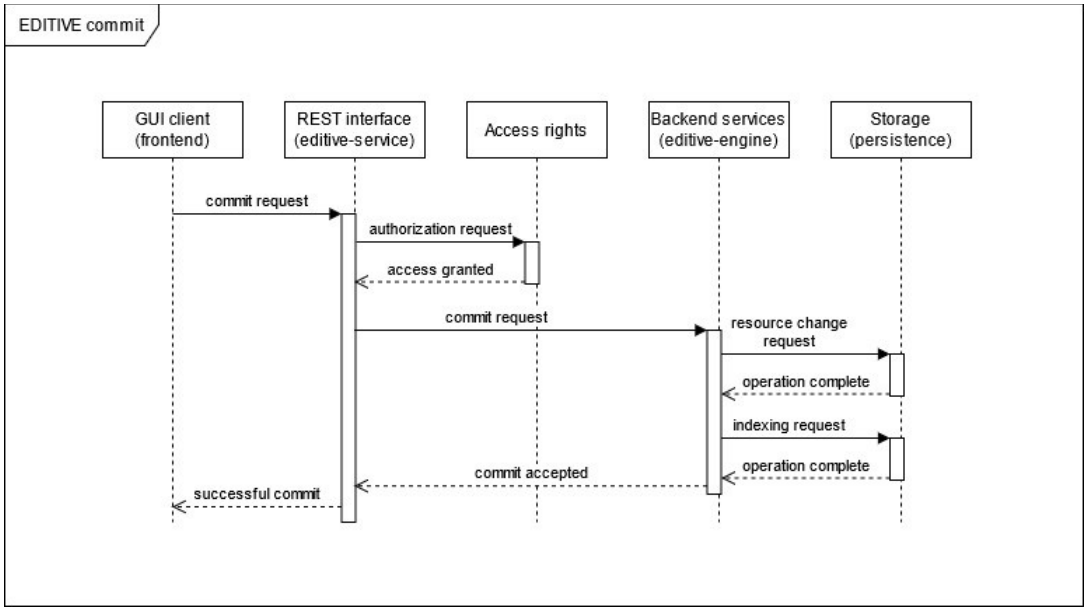

Figure 4.7: Sequence diagram for an EDITIVE commit

This sequence flow diagram shows a *commit request* from the *GUI client* to the *persistence* layer. The calls are issued sequentially in order to prevent the introduction of complexity through an asynchronous design in the current state of EDITIVE. The requests towards the *persistence* layer can be parallelized because the processing of both requests utilizes different resources underneath. There is no limiting time specified for the indexing operation, but it should not be more than two-fold the duration of the processing of the *resource change request*.

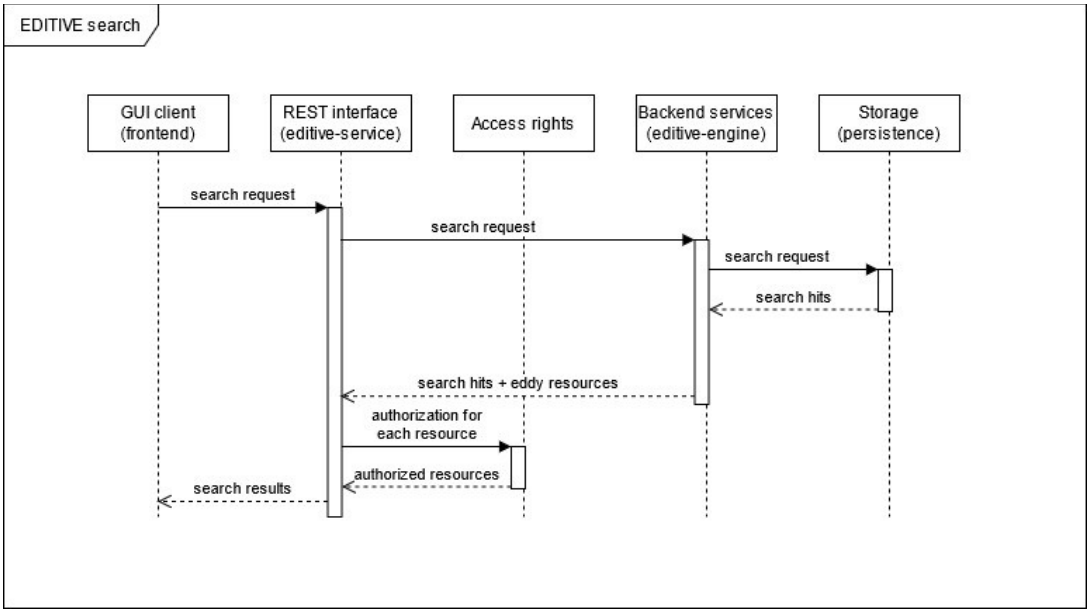

Figure 4.8: Sequence diagram for an EDITIVE search request

When processing a *search request* with a search index, the scale of access to EDITIVE resources depends on the amount of information stored in the index. If only IDs and data required for querying are stored, we need to later retrieve the objects from the EDITIVE storage, if a user wants to access these. An efficient way would be to retrieve enough information to display the hits to the user without retrieving all resources (Eddys and Eddy documents) found by the search. This information should include information necessary to validate authorization of the user, such that only accessible hits are shown. Currently, EDITIVE only provides authorization functionality within the *editive-service* layer therefore not allowing an earlier filtering of search hits.

4.1.4 Testing

All code should be testable. By utilizing the approach of a layered architecture, all parts should be unit tested. Furthermore, a possibility to test the implementation against a larger set of test data should be at hand. Currently, there is not a lot of test data available within EDITIVE in order to perform a meaningful performance test on the search implementation. Therefore, a small service is implemented to crawl through a Git repository, transforming all commits and files and inserting those into the EDITIVE application. This would enable performing manual and automated larger-scale tests on the whole backend system. Most Git projects contain code and, looking at the GitHub language rankings from 2018-2020, most code is JavaScript, Python and Java (GitHut, 2021). Therefore, this test is not focused on the content of the files but can be utilized to measure the indexing and searching performance of the search implementation.

Utilizing JGit, a simple implementation is done which traverses over and imports the commits of a Git repository. We can resolve the *HEAD* commit on a JGit repository and each commit offers a method to retrieve all parent commits.

```
ObjectId commitId = repository.resolve(Constants.HEAD);

RevCommit commit = walk.parseCommit(commitId);
```

Snippet 4.1: Retrieving the *HEAD* commit of a JGit repository

In this case, the method *resolve* on the *repository* object retrieves the *HEAD* commit ID. The *parseCommit* method on the *RevWalk* instance *walk* retrieves the actual commit object for a commit ID.

```
RevCommit[] parentIds = commit.getParents();
```

Snippet 4.2: Retrieving a commit's parent commit IDs

By executing method *getParents* on a commit object, all *parentIds* are retrieved within instances of the *RevCommit* class. But these objects only contain the ID, all other fields are null. The IDs from within these *RevCommit* instances have to be resolved in order to receive all *RevCommit* information. With these possibilities available in the JGit API, a simple *Git-importer-service* can be designed as follows:

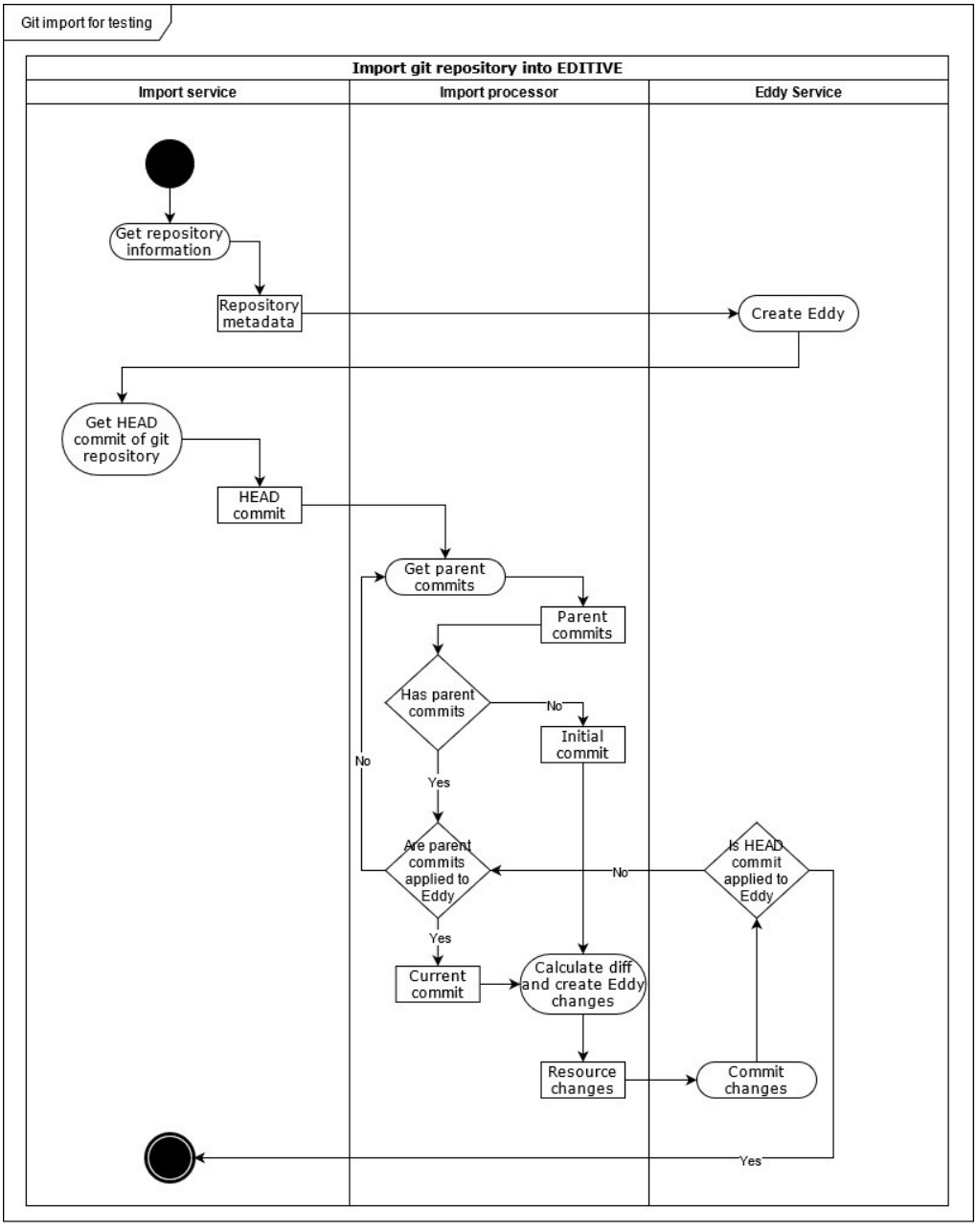

Figure 4.9: Git-importer-service import algorithm

First, a new Eddy is created from the Git repository metadata. From there, the *HEAD* commit of the Git repository can be retrieved. Next, we retrieve all parent commits and their parent comments by looping or traversing recursively. This executes until we reach the Git repository's initial commit. The initial commit is transformed to Eddy resource changes and these are committed on the Eddy. Now all commits retrieved earlier are transformed and processed following the "last in – first out" principle. If the *HEAD* commit is applied to the Eddy, the service can terminate and all commits from the Git project have been transformed and committed to the Eddy. With this simple *Git-importer-service*, commits and Eddy test data can be generated within EDITIVE.

4.2 Search functionality design

Solr provides a wide range of functionality, but not all functionality is required within EDITIVE. This architecture focuses on the indexing and searching of Eddys, Eddy documents, commits, authors and branches according to the requirements. Additionally, forks and commits have to be handled. The search functionality will only enable searching content on the *HEAD* commits within Eddys and will not perform a full historical search on all commits. Therefore, an efficient way of indexing and searching is designed.

4.2.1 EDITIVE actions

EDITIVE actions are initiated changes to content and content meta-information within EDITIVE. There are multiple actions within EDITIVE, which create data relevant for indexing.

- An Eddy is created.
- Eddy meta-information is changed (renames and changes of description).
- An Eddy is forked.
- An Eddy is deleted.
- A commit is performed on an Eddy.
- A branch or tag is created.
- A branch is renamed.
- A branch or tag is deleted.

If an Eddy is forked, the fork also contains all the existing documents. Therefore, all documents, branches and tags are required to be searchable for both resulting Eddys. Within EDITIVE, Eddy related resources don't receive new IDs.

All documents of the forked Eddy in the search index receive a field containing information on the Eddys they are referenced in. A **version reference** attached to Eddy documents can be introduced, which consists of an Eddy and a branch in its *HEAD* state or an Eddy and a tag.

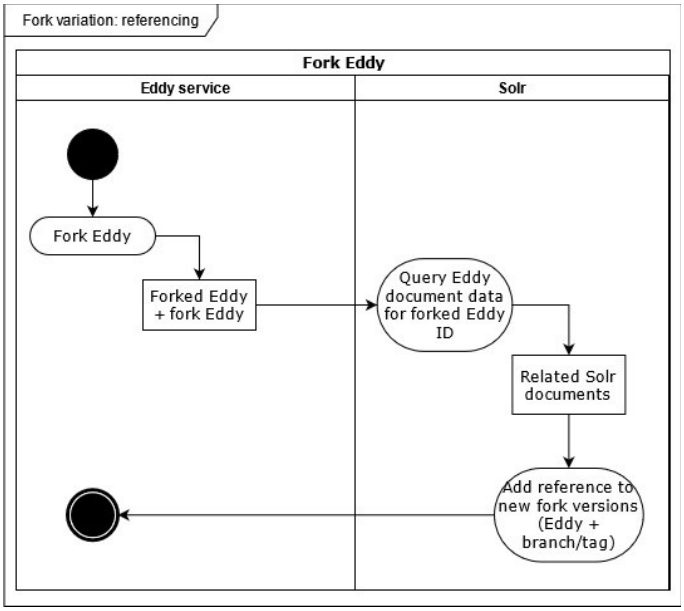

Figure 4.10: EDITIVE action: fork

This "referencing" approach is chosen because users are not expected to often change all resources within a fork. Instead, they are expected to be changing only a few resources after forking, tagging or branching. Regarding tags, it is possible that not all resources within a tag are changed. Therefore, adding the version reference to all relevant Solr documents is much faster and requires less indexing space compared to duplication.

An example of possible indexed data further illustrates the indexing behavior based on a "referencing" approach for all EDITIVE actions. Assume there is an Eddy named *Eddy1* with a document named *doc1* in its initial version named *versionA*. The Eddy has a branch called *master* on which changes to its documents can be performed. First, *Eddy1* is forked into *Eddy2*. The graphic below displays the new reference entry to *doc1* (*Eddy2:master*) in addition to its initial reference *Eddy1:master*.

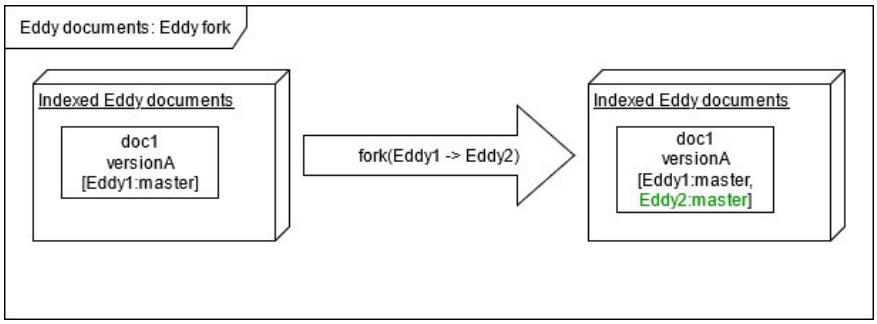

Figure 4.11: Index data example – forked Eddy

After *Eddy1* is forked into *Eddy2*, some changes are applied to *doc1* and *doc2* is added. These changes are committed on *Eddy1* as *versionB*.

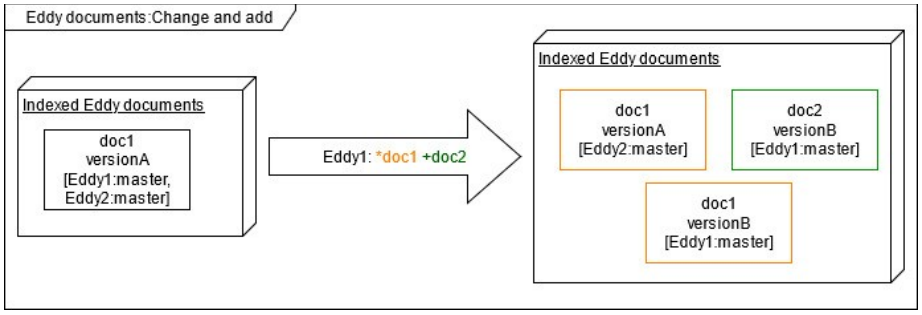

Figure 4.12: Index data example – * modify + add document commit

These changes result in three different Solr documents, representing two Eddy documents in different versions. Assuming a change is done within the Eddy fork *Eddy2* on *doc1*, the version for *doc1* referencing *Eddy2* would change. Since there are no other references than *Eddy2:master*, the result is still three documents.

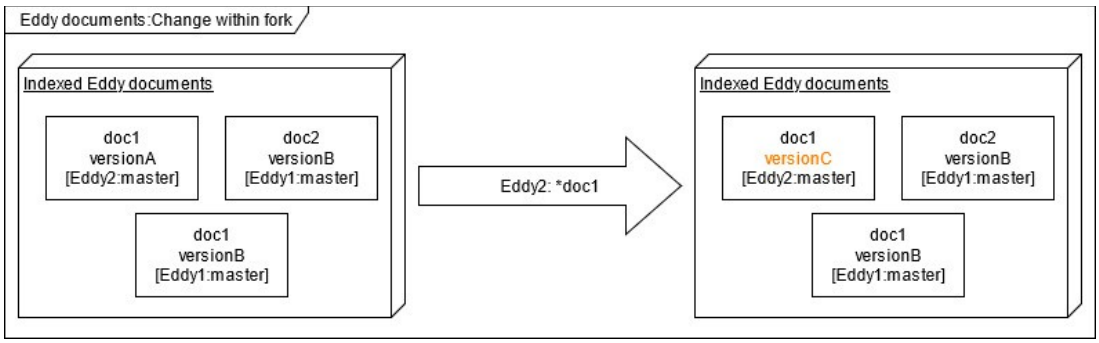

Figure 4.13: Index data example – * modify document commit

If a tag, *tag1*, within *Eddy2* is applied to *versionA* the result is the following.

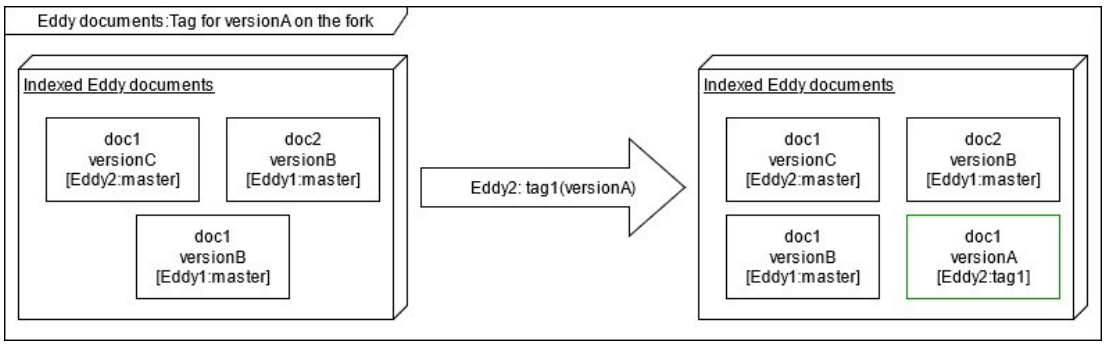

Figure 4.14: Index data example – tagging initial version

Tagging an "old" version of a document not known to the index leads to re-indexing of those version's unknown contents. Eddy documents within the index relevant for *Eddy2;versionA* would receive the reference to *Eddy2:tag1*. If tags are applied to the current version, no documents are added to the index. All relevant documents are indexed and receive the reference to *Edd2:tag1*.

Concluding, if a document loses all references, it can be deleted. This could happen if an Eddy, tag or branch is deleted. If the Eddy document itself is deleted, the reference of the deleted Eddy is removed. Branches are treated the same way as tags and represent a specific reference together with the Eddy. Collections and fields are designed utilizing this strategy to handle relevant EDITIVE action.

4.2.2 Collections and Fields

Solr's fundamental units of information are sets of data. These sets are called **documents** and should not be confused with EDITIVE documents. Solr documents are composed of **fields**. Every field has at least a name and a specific **field type** attached to it and can store data according to its field type. When receiving a document, Solr takes all information from the document's fields and adds those to an **index**. While querying Solr, it searches the index and returns matching documents. Another basic part of fields is the "field analysis" consisting of analyzers, tokenizers and filters. These are used to configure the handling of incoming data by Solr during the building of an index. "Field analysis" includes techniques such as stemming, removing stop words or configuration on how to handle upper and lower casing among many. In SolrCloud mode, we can define fields for a collection thereby implying defined fields for all documents within the collection. Two collections are defined to store relevant EDITIVE data. The collection *eddys* contains Eddy specific data. The collection *eddy_documents* contains Eddy documents specific data.

Fields in the collection *eddys*:

Name	**Type**
id	UUID
eddy_id	UUID
eddy_group_id	UUID
eddy_name	Text
eddy_description	Text

Table 4.1: *eddys* collection fields

The *id* field is a universally unique identifier (UUID) and is unique for each Solr document in the *eddys* collection. With an internal ID, internal logic can rely on that and is not dependent on external IDs. Within the *eddys* collection, the combination of *eddy_id* and *eddy_group_id* is unique, but for handling Solr documents the *id* field is used.

The *eddy_id* is an UUID and represents an Eddy within EDITIVE. Forks of an Eddy have their unique *eddy_id*. The *eddy_group_id* represents a group of Eddys. The initial Eddy and all its forks have the same *eddy_group_id* but a different *eddy_id*.

The *eddy_name* and *eddy_description* are simply text fields in which Solr's synonym filter and lower case filter can be applied. It could also make sense to apply a stop word filter regarding the Eddy description. The standard tokenizer is sufficient for both fields.

Fields in the collection *eddy_documents*:

Name	**Type**
id	UUID
eddy_group_id	UUID
eddy_document_id	UUID
eddy_commit_name	String
eddy_document_name	Text
eddy_document_content	Text
eddy_document_referenced_in	String

Table 4.2: *eddy_documents* collection fields

With this collection schema, there are only a few fields, which should be able to fulfill the requirements. The *id* field is an internal ID for each document in a version.

A version is defined by the *eddy_commit_name* where the secure hash algorithm string, identifying a specific Eddy commit, is indexed. This is used for the mechanisms, explained in chapter 4.2.1, and represents the version. Additionally, searches for a specific commit can be performed to find all documents changes within that commit or simply the commit itself can be retrieved.

The *eddy_group_id* is the same *eddy_group_id* as in the *eddys* collection, which can be used to narrow down the results of documents for an Eddy group.

The *eddy_document_id* is an ID from EDITIVE identifying an Eddy document.

The fields *eddy_document_name* and *eddy_document_content* are both text fields containing the name and the content of an Eddy document. The content of an Eddy document in EDITIVE is stored in the WOM format. The WOM is converted before indexing to a running text. Choosing the correct configuration for the *eddy_document_content* field improves performance, reduces storage usage and also accelerates the retrieving of search results, because the largest amount of data within EDITIVE is represented by the document content. Minimal English or German stemming can be enabled for indexing. Additionally, synonym filtering, stop word filtering and white-space tokenization can be applied to both, indexing and querying.

The reference to Eddys is performed in the field *eddy_document_referenced_in*. It is a "multiValued" field, which can be characterized as similar to an array providing high performance while adding and removing values, without changing other content of the Solr document. This enables simple addition and removal of references.

The author information is missing within the field definitions. A collection for author related information could be created, but author names don't require a lot of features provided by Solr such as word stemming. Additionally, the amount of data consisting of author information is small. Therefore, authors are searched for on EDITIVE's database and, in order to reduce complexity and indexing of author information, author data is not indexed.

4.2.3 Java interfaces for an implementation

Solr can be accessed via HTTP. Apache offers SolrJ as a small client application for Java, while other coding languages have to issue HTTP requests. We utilize SolrJ because it simplifies the code by using its client methods. SolrJ takes care of building the required HTTP requests.

Two interfaces are used to introduce layers of abstraction in the application. The first layer aims to enable the EDITIVE application to interact with a search functionality. It describes the requirements of the EDITIVE application towards a search functionality. The second layer utilizes the requirements of the search architecture and implementation specifically for Solr in the EDITIVE context.

To directly work on the repository, the *SearchRepository* interface is defined.

Figure 4.15: Java interface – *SearchRepository*

It contains methods required for searching, indexing (adding and updating) and deleting. Additionally, it provides the methodology for the use cases branching, tagging and forking. Depending on the implementation, these can be merged into one method.

The other interface is located in the service layer of EDITIVE, accessible by other EDITIVE backend services.

Figure 4.16: Java interface – *SearchService*

This Java interface is designed more towards the use within EDITIVE. It extracts information resulting from EDITIVE actions to provide relevant data for indexing. For searching, it restores familiar data types within the other *editive-engine* classes.

5 Implementation

This chapter describes the current implementation and possible future implementations for the EDITIVE team. In chapter 5.1, the Java implementation is described and some shortcomings are mentioned. Chapter 5.2 briefly explains how indexing is performed and chapter 5.3 outlines the querying.

All implemented code can be found on GitHub in the main EDITIVE project repository, previously on GitLab in the *editive-hub* project.

5.1 Java implementation and integration

The implementation is done directly embedded within EDITIVE. For a simple search functionality, some code is implemented using the existing technologies in EDITIVE. It enables searching for Eddy information such as description and name. Unfortunately, this solution is not sufficient for searching Eddy documents. Eddy documents are stored as binary data in the database. This data is retrieved from the database and serialized to WOM XML. This serialized WOM representation is kept in the random-access memory (RAM) and can be searched with Java string functions. If the search string matches content of a document, the document is retrieved and displayed as a search hit. This is performed on all documents within EDITIVE. Therefore, a majority of the database's overall data has to be read and processed utilizing Java string functions. This results in multiple performance issues and slows the overall system down. For that, we need a designated system capable of handling these kind of use cases. Therefore, we prefer the full Solr implementation while excluding author information. The resulting class diagram is displayed below. It focuses on the search functionality's implementation and excludes tests and test utilities such as the *Git-importer-service*.

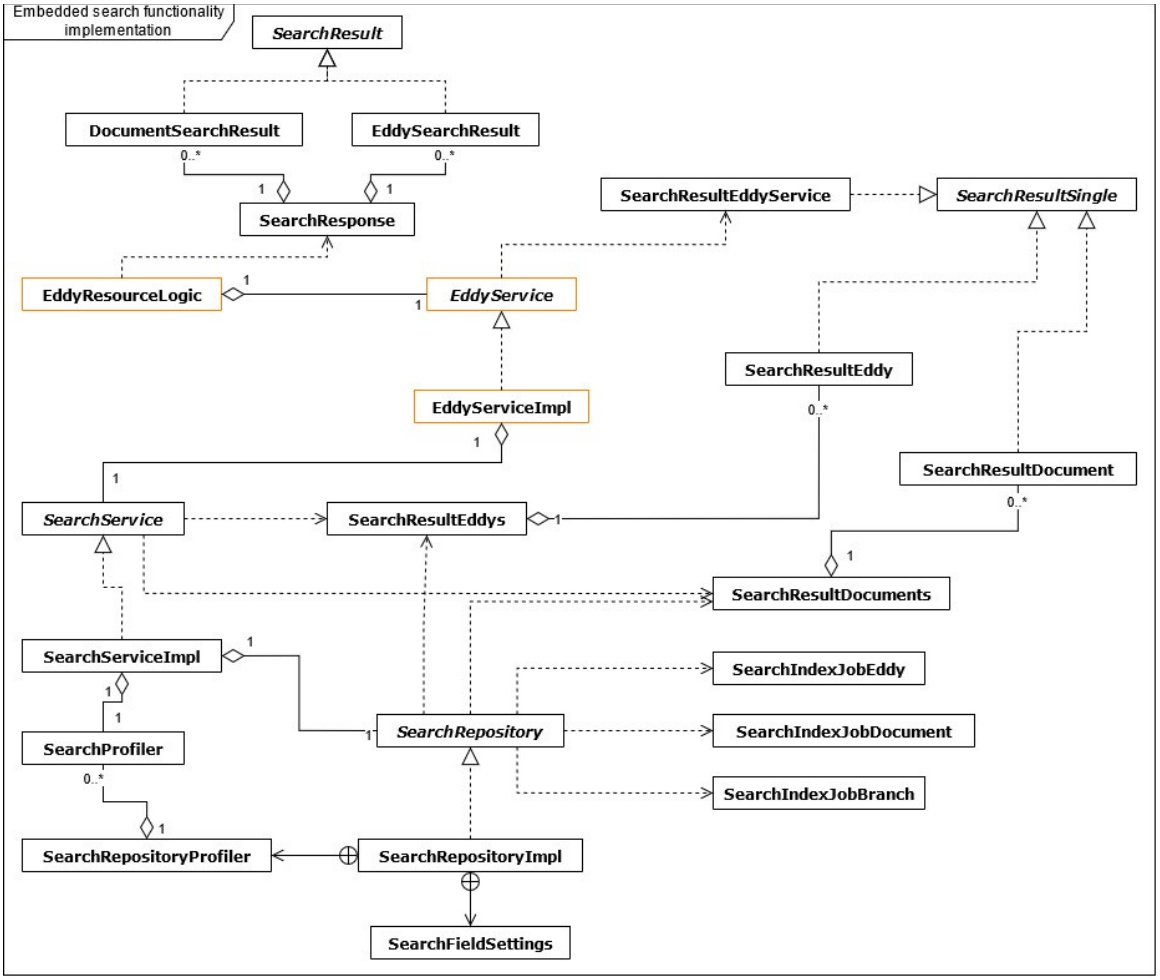

Figure 5.1: Implemented classes

Figure 5.1 displays the classes implemented during this thesis. The interface *EddyService* and the classes *EddyServiceImpl* and *EddyResourceLogic* existed beforehand but methods were added to these existing classes. A search method within the *EddyResourceLogic* represents the entry point for the search requests from the GUI.

The resulting implementation utilizes the *EddyService* for searching, because the *SearchService* itself doesn't contain all the information currently required for the GUI. The *EddyService* resolves all resources from the search hits, which are directly displayed and also authorized. This is not effective from an architectural point of view. The implementation following this is intended to be the first increment of the search functionality. Therefore, the most efficient solution is not implemented in order to prevent too many changes to the EDITIVE application such as providing user authentication on different objects and designing a new search hit list for the GUI. The EDITIVE team can be contacted regarding further implementation details. All code is available within a Git repository for authorized users.

The schema configuration is currently performed within SolrJ to ensure identical collections and fields within local development, during testing and on Kubernetes. The schema configuration is utilizing only the SolrCloud mode and not the standalone mode. The docker configuration uses Solr-slim containing only the minimal Java packages and thereby reducing the size of the image. This can easily be changed within the docker-compose file. The ZooKeeper is used as the environment necessary for SolrCloud mode even though only a single Solr instance is used within the docker-compose configuration as illustrated in Snippet 5.1.

```
search:
  image: registry.gitlab.com/editive-public/editive-hub/solr:8.5.2-slim
  container_name: editive-hub-2_search_1
  networks:
    editive-hub-dev:
      ipv4_address: 10.6.0.90
  extra_hosts:
    - 'local.editive.org:10.6.0.1'
  ports:
    - 3090:8983
  environment:
    - ZK_HOST=zoo1:2181
```

Snippet 5.1: docker-compose Solr configuration

Solr references ZooKeeper in its environment property and ZooKeeper then automatically manages all registered Solr containers. The number of shards for a Solr collection and the replication factor can be configured through SolrJ. The Solr nodes running on docker are configured within the docker-compose configuration.

```
zoo1:
 image: zookeeper:3.5
 container_name: zoo1
 restart: always
 hostname: zoo1
 ports:
   - 2181:2181
 environment:
   ZOO_MY_ID: 1
   ZOO_SERVERS: server.1=0.0.0.0:2888:3888;2181
 networks:
   editive-hub-dev:
     ipv4_address: 10.6.0.100
```

Snippet 5.2: docker-compose ZooKeeper configuration

To enable and disable the search configuration, a toggle switch is added to the *editive-service* configuration. This setting can be modified for unit and integration tests through test configurations. If the toggle is switched on for deployment or local execution, the code for the search functionality is then active and attempts to connect to an active Solr instance. If *isEnabled* is set to *false*, the toggle is switched off and the current implementation will return null values.

```
searchConfiguration:
  isEnabled: false
  host: local.editive.org
  port: 3090
```

Snippet 5.3: Toggle switch search configuration

Each implemented Java class containing testable logic is unit tested. Some integration tests are performed to assess the complete code behavior from the modification of Eddy documents to a retrieval of that information from the index. By utilizing Solr *testcontainers*, managing lightweight instances of Solr running in SolrCloud mode, unit tests covering the search functionality succeed within milliseconds.

5.2 Indexing

The indexing within EDITIVE is performed utilizing the SolrJ client. The Eddy references within the documents are implemented using a “multiValued” field. Currently, when there are no references remaining for a document, it is not deleted. It has no Eddy reference and therefore, when retrieved, it is identified as an obsolete document. This implementation of not deleting indexed documents is preferred for development and debugging reasons. Eddys on the other hand can be deleted explicitly through the *SearchRepository* Java interface.

The forking logic of Eddys and some of the branching logic have been implemented within Solr. A tagging logic has not yet been implemented. The tagging logic can be performed in the same way as the forking logic by using the referencing array within the Eddy documents collection. Since this tagging implementation follows the same logic as the forking implementation, it is not considered necessary regarding the evaluation of the search functionality’s efficiency.

The indexing implementation follows the architecture and is mostly performed within the *SearchRepositoryImpl* and the supporting classes. Logging for the *debug* and *trace* modes is included in the implementation. Additionally, extended logging, unit tests and integration tests are implemented including a few profiling points. These profiling points are switched off per default. This profiling measures the time consumption of indexing related actions interacting with Solr. It is separated in two steps. Step one includes all preparation for indexing and step two includes interaction with the Solr client by adding documents to the index and committing those. These time measurements can be used to further evaluate the code’s performance.

Field analysis is mainly configured through out-of-the-box field types. Solr’s field type *text_general* is used for the content of Eddy documents and no custom field type is configured. *text_general* utilizes the removal of stop words per default as well as the capability of processing synonyms and acronyms. It doesn’t provide stemming, since no language is specified. For basic English stemming, the *text_en* field type can be utilized.

5.3 Querying

All querying is handled in a similar fashion to the indexing through a SolrJ client. All queries, which can be entered by users, are first escaped to ensure security. This currently prevents the usage of operators within the queries, but should not be considered critical at this stage of the implementation. If further operator or keyword usage is required, search strings could be validated instead of escaping the query.

Queries are per default executed in both Solr collections, those containing Eddy information and those containing Eddy document information. The specific configuration to only search for Eddys or Eddy documents is accomplished by utilizing a Java *enum*. Currently, the GUI is searching within both configurations, but enabling the user to set the *EnumSet* can be added to the existing GUI controls.

In addition to the search hits, the implementation contains the highlighting of matching content as an example configuration of further Solr capabilities. The highlighting is enabled within the Solr query settings, resulting in content snippets from the matched documents containing the matching terms. The highlighting configuration allows the specification of fields, which should be highlighted within each query. The matching terms can be surrounded by configured strings to enable customized highlighting within the snippets as displayed in Snippet 5.4.

```
// define highlighted fields
solrQuery.set(QUERY_HIGHLIGHTED_FIELDS_PARAM, "*");

// define the max char length of the snippets
solrQuery.setHighlightFragsize(100);

// define if tags should be around the highlighted hit -> default is <em></em>
solrQuery.setHighlightSimplePre("");
solrQuery.setHighlightSimplePost("");
```

Snippet 5.4: Solr hits highlighting configuration

All highlights are flattened regardless of their fields. The format of the highlights returned by Solr is displayed in Snippet 5.5.

```
private Set<String> getHighlightsFromResults(
    final @Nullable Map<String, Map<String, List<String>>> highlights,
    final @NotNull String docId,
    final @NotNull String... fields
) {
  Set<String> snippets = new HashSet<>();
  if (highlights != null) {
    Map<String, List<String>> highlight = highlights.get(docId);
    if (highlight.size() > 0) {
      // flatten the highlights from list
      snippets = Arrays.stream(fields)
          .map(highlight::get)
          .filter(Objects::nonNull)
          .flatMap(Collection::stream)
          .filter(highlightEntry -> highlightEntry != null
              && !highlightEntry.isEmpty())
          .collect(Collectors.toSet());
    }
  }
  return snippets;
}
```

Snippet 5.5: Solr hits highlighting flattening

Solr returns highlights in a map containing all document hits and a map containing all fields and their matching lists of strings. The implementation flattens these maps for every document ID and all fields are passed as a parameter for flattening. This highlighting implementation is not achieved based on the proposed microkernel architecture, but could be extracted into a microkernel architecture when enabling further Solr capabilities.

The embedded querying of information is currently not as efficient as possible. The search hits from Solr are transformed and resolved utilizing the EDITIVE database. All Eddy resources are fetched from the database for the first page of search hits. Currently, the first page is configured to display up to 50 hits. This occupies a lot of resources on the system only to perform a later authorization on those resources for each search hit.

6 ISO/IEC 25010 evaluation

This chapter evaluates the architecture and implementation described in the previous chapters. The evaluation is performed, based on the evaluation scheme highlighted in chapter 2.3 *Evaluation scheme for requirements*, while utilizing criteria from the ISO/IEC 25010 standard for software quality. The category evaluation is performed independently for each criterion from the ISO/IEC 25010.

Functional suitability: Functional requirements have been implemented. A small search implementation was executed on the database before introducing Solr to the project. Whether the implementation facilitates the accomplishment of the user's tasks has still to be proven. It is expected to hold true according to the formulated requirements.

Performance efficiency: In order to analyze the performance efficiency of the application, a basic component has been implemented to measure time expenditure within searching. Five different operations are measured and the focus is set on their scaling. The five operations are:

- *index_prep*: The preparation of indexing including the processing of information from EDITIVE. Each operation represents collecting values from EDITIVE resource changes for commits.
- *index*: The indexing as executed by the SolrJ client. Every *index* operations executes a commit operation on the SolrJ client.
- *search*: The searching as executed by the SolrJ client.
- *unpack*: The unpacking of search results into a result format expected by EDITIVE backend services.
- *extract_commit*: The extracting of commit information required for indexing. Each operation represents an *add* or *update* resource change within EDITIVE.

The tests are performed on a local machine using the implemented *Git-importer-service* to generate test data. For these tests to provide meaningful results regarding scalability, they should be performed on a dedicated infrastructure instead. The tests on the local machine are influenced by other running applications. This test simply showcases the required duration for Solr to index test data. The test data is generated from the Microsoft Windows calculator project on GitHub. On the 2021-05-17, the calculator repository contains 654 commits and has a size of 32 505 kilobytes. The Solr instance as well as the EDITIVE service instance is running on the same machine, reducing the time of HTTP calls. Importing this repository ten times into different Eddys and performing a subsequent search, the duration in table 6.1 is measured.

Measure	**Total duration in nanoseconds**	**Total operation count**
index_prep	1 099 947	71590
index	12 495 980	71590
search	38 845 588	1
unpack	341 054	1
extract_commit	106 100	71580

Table 6.1: duration of search functionality during ten Microsoft calculator imports within EDITIVE

The discrepancy between the number of *index* operations and *extract_commit* operations can be explained by the number of ten indexed Eddys. This test showcases the measures of the simple profiling implementation. Additionally, it validates the performance efficiency of the

implementation even though it has a high margin of error. Taking 38 ms, the searching of relevant resources is the longest. The indexing of data is performed within less than a third of that time.

Compatibility: Integration and unit tests are implemented covering the search functionality. All components of the search functionality are directly integrated within the EDITIVE backend. All Java interfaces make use of EDITIVE data types. Changes to the code passed successful deployment builds and were merged through a Git merge request from a fork repository.

Usability: User protection is established by escaping search strings. The search bar is simple to use and currently provides no additional possibility of configuration. The Java interfaces are easy to use and utilize the EDITIVE internal data types.

Security: Only basic security measures are applied. All user access and project visibility settings are checked before transferring any information from the backend. Additionally, search strings are escaped to prevent harmful modification or unintended access to indexed data.

Maintainability: The implementation is mainly split in two parts. One part is the adapter towards any existing EDITIVE functionality, the second part abstracts the Solr specific implementation. This enables switching between or comparing different search platform implementations. Additionally, most code is covered through automatic tests, commented where necessary and basic logging is performed. The code is structured similarly to the EDITIVE architecture and is therefore easier to maintain by EDITIVE team members.

Portability: The portability of the search platform implementation to a different content collaboration system than EDITIVE is not straight forward. The architecture can be used across multiple platforms, but the implementation is EDITIVE specific. The adapter on top of the search platform implementation should be portable to different search platform logics implementing the *SearchRepository* interface.

7 Outlook

The implementation fulfills its requirements concerning the handling of forks and versioning, utilizing commit data. EDITIVE enables the user to create tags, which version's contents should be searchable as well. The handling of tags can be implemented similarly to the forking logic implementation.

Several Solr features have been implemented to provide further features during querying and indexing. But for future users this might not be sufficient. Several Solr features could be used within the proposed microkernel architecture. Faceting can be implemented if required by users. The Java interfaces might require to change for that implementation, but can still rely on the classes used for the return values and method parameters of the search implementation. These result and parameter classes can be extended without changing the Java interfaces. Another feature to optimize the search functionality by is pagination. The current implementation only displays one page containing up to 50 search hits ordered by their relevance. Additionally, a text representation of Eddy documents can be further detailed. The XML support of Solr can be utilized to quickly index Eddy documents in a WOM representation. The requirement to enable performing searches, restricted to document titles, abstracts or the whole document, was discarded in order to focus on the more difficult implications of a search functionality within EDITIVE. The WOM documents can be indexed without applying major changes to their structure utilizing Solr's XML support. The result could be compared to the performance of an existing WOM search implementation (Miller, 2018).

The current implementation ensures, that user access rights and project visibility settings are taken into account. This leads to the resolving of all resources from search hits within the current EDITIVE implementation. It can be further optimized by only loading a resource after a user selects the corresponding search hit. User authorization should be validated beforehand, based on the resources' IDs retrieved from the search index. Additionally, the *EddyServiceImpl* class uses the *SearchService* interface for searching. The *ResourceLogic* class could instead directly access the *SearchService* interface. Currently, this implementation is necessary because of the system's conditions.

The search implementation could be tested against the proposed search architecture of Robert Miller (Miller, 2018). An implementation could be added as a separate *SearchRepository* instance utilizing Elasticsearch. By utilizing the *Git-importer-service,* a sufficient set of test data can be generated. Additionally, the generated test data can be used to further validate the work from Robert Miller as well as this implementation independently.

The logic for the *Git-importer-service* is functional and sufficient to generate test data for EDITIVE. The expenditure of resources has to be considered when designing the algorithm. By utilizing recursion, objects within each recursion could be kept in memory, therefore resulting in a stack overflow depending on the program logic and hardware it is executed on. The *GitServiceImpl*, responsible for the import of Git projects, utilizes a recursion strategy based on the available JGit interface and is unable to import *torvalds/linux* into EDITIVE on a desktop computer due to the large size of data and the large number of commits.

8 Conclusion

In this thesis the development of a search functionality for the multi-level content collaboration platform is presented. The development of this functionality is not complete and remains a task for the future. The presented implementation provides a search functionality on the *HEAD* commits of EDITIVE projects. It doesn't distinguish between document titles or abstracts but always treats the document content as one field. It is embedded within the EDITIVE backend and respects the project's visibility settings and the user's permissions. The resulting implementation includes a utility to generate test data and a basic logic to measure performance of the search functionality. The evaluation of the requirements verifies the implementation.

This work outlined the importance of requirements for an architecture and implementation. An evaluation scheme was proposed to verify the resulting architecture and implementation according to its requirements. The categories for the evaluation scheme are chosen from the ISO/IEC 25010 standard for software quality. Subsequently, the currently popular technology Apache Solr and the reason behind choosing it for this thesis's implementation are described. The proposed reference architecture, its implementation and evaluation are described. Based on architectural patterns we present an architecture for the search functionality. The handling of the commit-based workflow within EDITIVE represents the greatest challenge including forking, branching and tagging. Additionally, the importance of interoperability of software is stressed. Subsequently, the evaluation is performed based on the evaluation scheme described within the first part of this thesis. Lastly, we propose several future development cases, including a Solr configuration, facilitating the functionality for the user, as well as optimizations to the current implementation.

There is some room left for further optimization of a search functionality within EDITIVE. Apache Solr as a power search platform facilitates an efficient implementation for many use cases. The main challenges were the integration within the EDITIVE and the reduction in complexity of the collaboration workflow of EDITIVE to enable an efficient search.

All implemented code can be accessed with appropriate rights at the EDITIVE project in Git.

Appendix A Bill of Materials

Artifact	Version	License	Date last checked
Apache Solr	8.5.2-slim	Apache 2.0	2021-05-08
Apache Solr SolrJ	8.7.0	Apache 2.0	2021-05-09
Apache ZooKeeper	3.5	Apache 2.0	2021-05-08
Dagger	2.30	Apache 2.0	2021-05-09
Jackson core	2.11.3	Apache 2.0	2021-05-09
Jakarta inject	1.0.3	EFSL v1.0	2021-05-09
Jakarta validation	2.0.2	EFSL v1.0	2021-05-09
Jetbrains annotations	20.1.0	Apache 2.0	2021-05-09
slf4j	1.7.30	MIT	2021-05-09
Testcontainers Solr	1.15.1	MIT + Apache 2.0	2021-05-09
JGit	5.9.0	EDL v1.0	2021-05-09

Table 8.1: Bill of materials

References

Abdullah, Z. H., Yahaya, J. H., Mansor, Z., & Deraman, A. (2017). Software Ageing Prevention from Software Maintenance Perspective–A Review. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, *9*(3-4), 93-96.

Aliannejadi, M., Zamani, H., Crestani, F., & Croft, W. B. (2018). Target apps selection: To wards a unified search framework for mobile devices. *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval,* 215-224.

Apache. (n.d.). *Welcome to Apache Solr – Apache Solr*. https://solr.apache.org

Dohrn, H., & Riehle, D. (2011). Wom: An object model for wikitext.

EDITIVE. (n.d.). *Enterprise Multi Level Content Collaboration und Diff – EDITIVE*. https://editive.com

GitHub. (n.d.). *Search – GitHub Docs*. https://docs.github.com/en/rest/reference/search

GitHub. (2017). *Github search sucks (and how it could be better)*. https://github.com/isaacs/github/issues/908

GitHut. (2021). *Github Language Stats*. https://madnight.github.io/githut/#/pull_requests/2021/1

Gupta, D., & Khanna, A. (2017). Software usability datasets. *International Journal of Pure and Applied Mathematics, SCOPUS*, *117*(15), 1001-1014.

ISO. (n.d.). *ISO 25010*. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Lohr, S. (2012). For impatient web users, an eye blink is just too long to wait. *New York Times*.

Lown, C., Sierra, T., & Boyer, J. (2013). How users search the library from a single search box. *College & Research Libraries*, *74*(3), 227-241.

Miller, R. (2018). A search for Sweble Hub – Indexing structured data for a commit based search.

More, N. T., Sapre, B. S., & Chawan, P. M. (2011). An insight into the importance of Requirements Engineering. *International Journal of Internet Computing*, *1*(2), 34-36.

Niu, N., Da Xu, L., & Bi, Z. (2013). Enterprise information systems architecture—Analysis and evaluation. *IEEE Transactions on Industrial Informatics*, *9*(4), 2147-2154.

Office of Public Affairs (2020). Justice Department Sues Monopolist Google For Violating Antitrust Laws. *Justice News*.

Robillard, M., Walker, R., & Zimmermann, T. (2009). Recommendation systems for software engineering. *IEEE software*, *27*(4), 80-86.

Roshdi, A., & Roohparvar, A. (2015). Information retrieval techniques and applications. *International Journal of Computer Networks & Communications Security*.

Zhou, S., Cheng, K., & Men, L. (2017). The survey of large-scale query classification. *AIP conference proceedings*.