Hochschule
Augsburg University of
Applied Sciences

Masterarbeit

Fakultät für
Informatik

Studienrichtung
Informatik

Felix Kampfer

Knowledge Base Supported Document Retrieval and Annotation

Erstprüfer: Prof. Dr. Wolfgang Kowarschick
Zweitprüfer: Prof. Dr.-Ing. Hon. Dr. of ONPU Thorsten Schöler
Abgabe der Arbeit am: 20.09.2019

In Kooperation mit Firma:
Bayerische Motoren Werke Aktiengesellschaft
Petuelring 130
80809 München

Betreuer: Heinrich Maier

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon +49 821 5586-3450
Fax +49 821 5586-3499

Verfasser der Masterarbeit:
Felix Kampfer
Baumkirchner Str. 15
81673 München
Telefon +49 152 0483 9964

felix.kampfer@gmail.com

Hochschule
Augsburg University of
Applied Sciences

Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

_____ _____

Ort, Datum Unterschrift des/der Studierenden

Contents

Abstract

In order to help employees of BMW's TWA department benefit from the results of previously completed analyses, a custom-made system to find the most similar reports for a given work order was successfully implemented. The prototype employs a unique combination of modern web technologies, graph-based data management, and state-of-the-art natural language processing methods to deal with the challenges of uncategorized report content and inconsistent report structure.
An evaluation of the system's suitability as a productively-used internal tool is still ongoing.

Keywords: knowledge base, document annotation, document retrieval, information retrieval, NLP, text similarity, React

Glossary

The following table provides a quick overview of the terms used in this document.

Term	Meaning
ADM	The ADM (Document Management) database containing the metadata of all reports and orders including author and filepath. The database is considered to be an external component of the system.
Analyst, lab technician, lab worker, technician	Used interchangeably to refer to employees of the TWA department who fulfill damage and/or quality assessment requests by conducting visual, physical, chemical and other analyses on lab samples.
(Car) Component	Any physical thing, or treatment applied to a physical thing, or system that makes up part of an automobile.
(React) Component	A JSX-based class or function combining UI logic and markup instructions.
JSX	Javascript Extended, a Javascript-enabled HTML templating language for use with React.
(Work) Order	An assignment to assess the extent or investigate the cause of damage to a component, or to verify a component's adherence to certain technical standards.
Report	The written documentation of a completed work order, usually containing a list of the related samples, the analysis to be done (or the properties / qualification to be investigated), a presentation of the experimental results (often with pictures), and a conclusion and recommendation paragraph.
TWA	The "Technology Materials and Process Analysis" (Technologie Werkstoff- und Verfahrensanalytik) department of BMW, responsible for assessing the quality of components processed by internal and external sources.
TWADA or Twada	The prototype software presented in this thesis (TWA Data Analytics).
User	Any person interacting with the web application (using a browser)
Website	The [frontend] web application of the system that the user interacts with, hosted by the server.

Chapter 1

Introduction

1.1 Goal

As part of this thesis, a prototype of a web application to search, compare, and retrieve investigative reports in a technical domain was successfully implemented. The application was designed for use by the BMW Group's TWA (Technology Materials and Process Analysis) department, responsible for verifying the properties of components purchased from suppliers and for conducting investigations of damaged components in order to determine the nature and source of the damage. Each of these analytical tasks is initiated by the creation of a work order by a client and is concluded by the creation of a report by an analyst. Due to the high volume of orders and reports (up to 100 incoming daily orders, with 400,000 archived reports to search through), it is possible for certain investigative efforts to take place more than once, wasting valuable machine and human resources in the process.

The purpose of the system was to enable TWA employees to find existing reports matching the subject matter of an incoming work order, thereby allowing them to benefit directly from the conclusions reached by other analysts, such as the kinds of chemical or mechanical tests that can offer conclusive answers to questions about damage type.

1.2 Context

1.2.1 Available Tools

TWA employees have several internal research tools at their disposal, including the "Wissensfinder", a web-based document search application, "WinLIMS", a desktop client for managing laboratory information in ADM, and "IMagic", a browsable database of images used in the department's lab reports. While these tools allow employees to look up previous reports, they are dependent on the metadata of the reports, such as the title field, or the vehicle model field. This

metadata in turn is dependent on the quality of information provided by submitters and task handlers, which varies. Even when a report's metadata is accurate, it can be insufficient for later analysis. This is partially a result of immutability conventions, meaning that the title or description of an order is never updated to match the results of an investigation. Additionally, the results of investigations are only available in Word and PDF formats, the contents of which the existing tools of the TWA can't programatically parse.

1.2.2 Organization of the Department

The TWA's is split into six organizational units comprising an administrative group (dealing with IT, planning, and the development of new internal tools such as TWADA), and five so-called centers of competence (COCs). These COCs each have the know-how and expertise of one particular area of analysis, including metal (responsible for e.g. screw technology, general workpiece quality analysis, fire prevention), nonmetal (e.g. durability analysis of textiles, leather and polymers, as well as automobile acoustics analytics), chemistry (e.g. analysis of process materials and emissions), batteries (e.g. analysis of hybrid systems, prototype battery cells and high voltage systems), and surface materials.

1.2.3 Previous Work

This thesis builds on the results of previous work by another student employed by the TWA, which explored format-agnostic text extraction and experimented with multiple machine learning-based classification algorithms of the TWA's reports. While we make use of findings dealing with text extraction, the classification algorithm and especially the presentation of results used in the current system is different.

1.3 Approach

The main goal of the system is to reduce the amount of "guesswork" involved in investigating the source of damage to certain car components. As part of development of this system, work was split between development of a backend system that can process a request to find similar reports, and the development of a frontend system that allows users to send such a request and see a list of similar reports. After early agreement with lab analysts regarding the general structure of the web application, the webserver for the frontend was merged with the API server into a single backend system. When the critical features of the system were implemented, work began on developing the annotation editor and connecting it to the system's node-based database.

1.4 Results

The prototype successfully fulfills all critical and desired functions, allowing lab workers to find similar reports and annotate existing reports using a single modern web application. Evaluations of the software have been conducted by five analysts, with largely positive feedback. Further evaluation is necessary to determine if the system is an ideal candidate for productive use as an internal TWA tool.

1.5 Outline of This Thesis

The body of this thesis is split into three major parts, outlining the application's technological foundations, task requirements, and implementation details. Chapter five summarizes the results of this thesis and evaluates how well the system accomplishes its goals in greater detail. The final chapter gives an outlook on opportunities for further development of the application in the realm of application development and data mining.

Chapter 2

State of the Art

This chapter introduces the theoretical and practical foundations of the application, split into the titular aspects of document retrieval and annotation, as well as relevant elements of web application development and deployment.

2.1 Document Retrieval

Document retrieval (also referred to as information retrieval) is the science of storing knowledge and answering queries. It involves a large variety of disciplines including information theory, machine learning, probability theory, computational semantics, NLP, algebra, logic, and others. Research is also ongoing in areas of data presentation, storage (including indexing, encoding and compression), classification, and systems architecture with special attention paid to issues such as performance and space efficiency. [2]

Information retrieval systems have their roots in the automated indexing of libraries' text sources with the support of Hans Peter Luhn's work on automatic keyword assignment.[2]

Structured Document Retrieval

A subcategory of document retrieval is structured document retrieval, or passage retrieval, wherein specific parts of a document are used to satisfy queries (either by returning the matching passages or the entire document). These passages can be found within manually annotated sections of a document (such as within a markup language) or they can be derived via algorithmic topic segmentation. [**infoRetrievalSlides**] In scientific articles, for example, text is found in the abstract and in the body, which is split structured into sections and subsections containing paragraphs. While structured document retrieval is different from traditional document retrieval, which only deals with a single "raw text", the topic has come to be called "semi-structured" document retrieval to differentiate it from database technologies that deal with highly structured fields such as authors and dates. [16]

Search Engines

Internet search engines present a prominent example of document retrieval in everyday life, comparing user queries to documents found on the web and returning an ordered list of results. Web search engines create indexes of websites by crawling through petabytes of content hosted on the web, and can respond to millions of parallel queries within fractions of a second. Other features of search engines include the multi-faceted interpretation of user input, and the ability to mine data from databases. [7]

Google is the most well-known web search engine in the western hemisphere, originally presented by Lawrence Page and Sergey Brin in 1998 [**google**]. It came to fame partially as a result of its PageRank algorithm, which retrieved documents for users in a special order, namely by "effectively measuring the human interest and attention devoted to them" based on the number of sites linking to a certain document, and the number of sites linking to the linking sites. [21].

2.1.1 Similarity Identification

An important subfield of information retrieval is the determination of similarity. The field of natural language processing can help with this problem, concerning itself with the semantic processing of spoken and written text.

The field of natural language processing is large, concerning itself with a large variety of language-related challenges. These include multilingual tasks (spelling, translation), information extraction (question answering, text summarization or simplification), emotional analysis (sentiment analysis, opinion mining and irony/sarcasm detection) and spoken language understanding. In order to be able to help analyzing the semantics of a given text or document, it must be broken down into simpler elements. A data scientist's proverbial NLP toolchain therefore includes ways to segment sentences, tokenize individual words (optionally with the help of part of speech tagging) and ways to derive words' morphology.

2.1.1.1 Comparison via Word Frequency

As detailed in the implementation's backend section, the implemented document retrieval system uses a multi-step process to identify reports that contain similar contents as a given search object. After extracting, pre-processing and classifying the input document's text, the system uses a bag of words approach (see [27]) to compare the document's contents with the contents of all documents in the corpus.

$$\mathrm{tf}(t, D) = \frac{\#(t, D)}{\max_{t' \in D} \#(t', D)}$$

Figure 2.1: Term Frequency

Specifically, the system makes uses of the so-called TFIDF score, which can be used to evaluate the importance of a word in a document compared to the corpus. TF refers to term frequency, and measures the relative frequency of a word in a document. If a word occurs more frequently than other terms in in a document, then this word is theorized to be of greater importance (see Figure 2.1).

$$\mathrm{idf}(t) = \log \frac{N}{\sum_{D:t \in D} 1}$$

Figure 2.2: Inverse Document Frequency

IDF refers to inverse document frequency, pitting the importance of a word in a single document to the prevalence of these documents in the corpus. Specifically, IDF is calculated by taking the logarithm of the quotient of the total number of documents and the number of documents containing the term in question (see Figure 2.2).

$$\mathrm{tf.idf}(t, D) = \mathrm{tf}(t, D) \cdot \mathrm{idf}(t)$$

Figure 2.3: TFIDF Score

In order to compare the semantic contents of documents, the vector of the TFIDF scores for every word of a particular document is compared to the same vector for every other word (see Figure 2.3). This comparison is achieved via the Cosine similarity of the vectors. A two-dimensional similarity matrix is generated as a result of this procedure, offering instant lookup of similar documents for any TFIDF-vectorized document [22].

2.1.1.2 Comparison via Word Context

Many other methods to compare the contents of documents exist as well. While TF-IDF uses a ”bag of words” approach to compare semantic meaning, other approaches deal with so-called word embeddings, studying the context of words, phrases and even paragraphs to derive similarity vectors. These vectors describe the words or phrases that pre- or append the term being evaluated. [10]. This approaches is also feasible for entire documents in the form of the doc2vec algorithm. [17].

Lemmatisation and Stemming
In order to semantically compare the contents of documents, it is helpful to transform words to their root form by removing factors such as conjugation, plurality and tense. By doing so, similar subject matters can be detected regardless of grammatical context. Lemming and stemming are two ways to achieve this grammar-independent state. Stemming is the more naive of these two variants, and works cutting off the last several letters of a word if they appear to be a common suffix. The shortened word is then treated as the root of the original word. Lemming on the

other hand involves using an existing word bank to detect the original word that an expression was part of, and returning the original word. This variant is only possible if all of the input words are contained within the word bank, which is difficult in domains with a lot of technical words that are not part of the language's normal corpus. Stemming is therefore the only feasible solution for scenarios with language ambiguity in applications with data sparsity (such as for low-resource languages, high morphology languages (such as Finnish), non-standard spelling, or domain-specific texts).

2.1.2 Challenges in Comparing Reports

The nature of its work, its decentralized organizational structure, and the standards for its reports present many challenges for the processing of TWA reports in an automated way.

Challenges of Domain-Specific Texts
Domain specific terms, such as names of technical examinations typical for the automobile industry or descriptions of internal car components, do not exist in the standard word banks of NLP libraries for a given language. As a result, part-of-speech taggers have problems correctly identifying some of these words and attempts at algorithmically deriving information from a selected text may be fruitless. Even attempts at deriving or annotating information in such documents by humans may be non-trivial, as specific domain knowledge may be necessary to interpret the language used. Similar challenges exist for developers creating tools for highly technical domains.

Challenges of Decentralized Departments
Another challenge for automated processing of reports is the tendency for isolated or separated teams to develop different vocabulary to describe the same subject matter, making semantic comparisons more difficult. A similar tendency is the independent creation and adaptation of norms and data representation among separate groups, such as geographically or organizationally separate centers of competence. In other words, the decentralized nature of the TWA's COCs can create subtle differences in syntax and semantics. These findings were made during early development of the similarity identification algorithm, wherein some reports were often calculated to be similar to other reports with a different subject matter but the same author, who used the same kind of language language to describe his work and his results.

Challenges of Formless Texts
A third challenge for the machine readability of texts is a lack of predictability of the report contents. While all German reports that we have encountered so far did adhere to the given structure of "Vorgang", "Ergebnisse" and "Folgerung" (sometimes "Empfehlung"), the freeform nature of the sections' contents necessitated the use of inexact natural language processing techniques to find semantic similarity (see Section 4.3.2.4 about specific problems with the "Ergebnisse" section). If a limited set of possible response formats were to be mandated, including the requirement

of adding normalized metadata such damage type and component name to new reports, then the task of finding similar reports might be a trivial one. We predict that such changes would not be welcomed by all lab technicians, however.

2.1.3 Existing Software Solutions

ElasticSearch, one example of an existing document retrieval system, is a document search engine based on Lucene, enabling rapid full text search on large corpuses. One of its strengths lies in its ability parallelize the real-time indexing of documents using multiple connected machines. ElasticSearch's API offers the ability to find similar documents (using a bag of words / TF-IDF-based approach) via its MLT ("more like this") query. This functionality encompasses extracting keywords from a search query (or input document) and scoring the terms based on importance before executing the query. ElasticSearch also offers a number of alternative similarity models such as BM25 (TFIDF optimized for short fields), LMDirichlet (Bayesian smoothing of a multivariable probability curve) and LMJelinekMercer (capturing important patterns in a text while minimizing noise). [5]

2.2 Document Annotation

The word annotation by itself can refer to any kind of metadata associated with digital or nondigital data, including maps, forum posts or any kind of text document. In the context of software engineering, annotations can refer to the usage of markup languages such as XML and HTML to give some raw text additional information regarding how to render it, or to indicate machine-readable information for web crawlers and other software.[24] Machine readable information on the web forms the basis of the semantic web, a set of standards by the World Wide Web Consortium (W3C) describing how machine readable information (using agreed upon definitions of entity types and relationships) should be annotated online. [4]

2.2.1 Knowledge Management

Knowledge management is the broad field of organizing, storing, and regulating access to both raw information and derived awareness (knowledge). While most types of databases have the ability to store virtually limitless amounts of data, so-called knowledge bases represent a particular class of databases that can store structured data with references to other complex structures of a particular class or subclass in a highly customizable format. While SQL databases are limited to formats such as strings, numbers, dates, and perhaps geo-based data, knowledge bases can define hierarchical types of knowledge and allow greater interconnectivity of data while tolerating faulty information. [11]. An ontology describes the types and relationships used in a knowledge base. An ontology may have certain inference rules that describe a set of conditions that must be met to reference an existing set of data using a certain relationship or property.

2.2.2 Existing Software Solutions

Multiple software solutions exist in the realm of knowledge management. So called triplestore, or RDF (resource description framework) databases are a specialized form of graph databases with nodes and edges to store information, as well as attributes as part of nodes and edge. Collections of well-known graph database software include neo4j, OrientDB, Virtuoso and various implementations of the SPARQL query language. In terms of deriving semantic information from texts, the Java-based GATE library (General Architecture for Text Engineering) features a number of modules (including named entity recognition and coreference detection) to process textual information in documents in order to automatically create a machine-readable representation of knowledge that is searchable. [1]

2.3 Web Application Development

2.3.1 Frontend Frameworks

Web application frameworks (WAFs) like React enable developers to develop applications with reusable parts including visual designs, data structures and independent data processing components.

The most popular web frameworks at time of writing (measured by the stars on their respective GitHub pages) are React, Vue and Angular (together with Angular.js, Version 1 of Angular).

Popularity has no bearing on quality or suitability for any particular product, however the large number of developers using (and helping develop) these frameworks ensures that a large amount of community support exists, especially in the form of tutorials, asked and answered questions, and third party extensions. Furthermore, it will be easier to continue the project from a software development standpoint, as more developers will have the relevant skills to work with the framework.

[25] details criteria for choosing a web framework, split into the categories of the framework's ecosystem (maturity of the framework, popularity of usage, corporate support or other sponsors) tooling (existing libraries for UI and logical components, IDE extensions, CLI tools), enterprise options (license, enterprise support options), and framework structure itself (tutorial and learning curve, required skills to get started, completeness of features, performance considerations, and mobile support).

Of particular praise are Angular and React's recent big refactorings which serve as an indicator of maturity and stability, despite serving as a warning of ongoing breaking changes. React also scored points for its increased adoption rate (React has 5 million weekly NPM downloads compared to Angular CLI and Vue having 1 million each), with notable brands including Facebook, Netflix, Paypal and AirBnB. [25] notes that Angular and React have Google and Facebook as serious corporate sponsors, although warning that corporate entities have abandoned large projects

in the past. In terms of framework structure, Vue and React are seen as more lightweight and easier to learn, while Angular has a more complete feature set (i.e. having its own routing logic).

In the end, React was chosen to be the framework of choice for TWADA's frontend, in part due to positive personal experience.

2.3.2 Interface Design

2.3.2.1 UI Patterns

UI patterns are reusable interface concepts for receiving input and giving output from users. In terms of input, patterns such as input prompts and inplace editors allow for the rapid querying of information and modification of existing data. Such inplace editors may include textboxes (or selection boxes) that allow users to select or create tags to label and categorize documents.

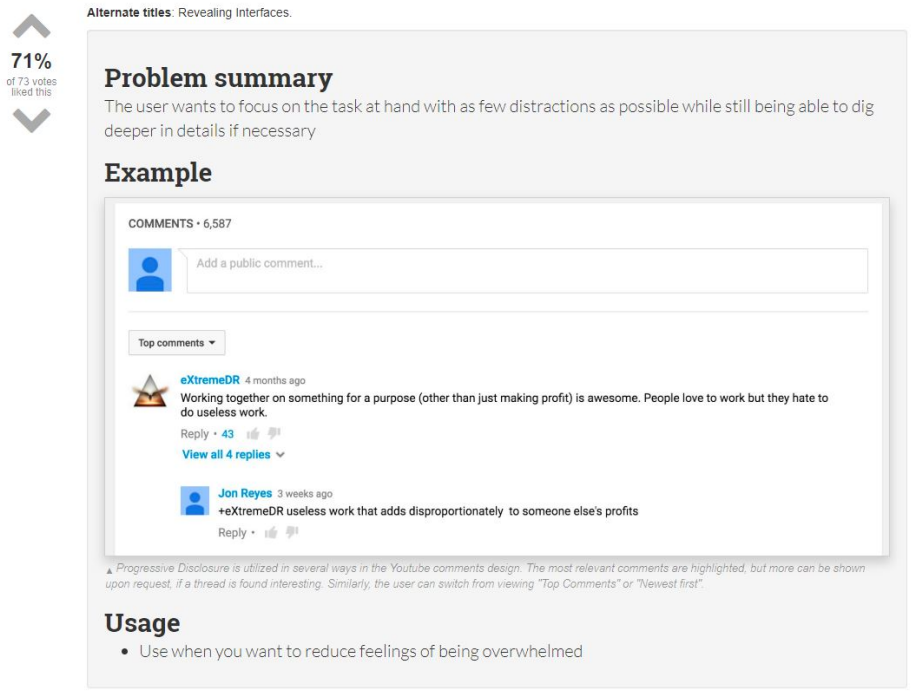

Figure 2.4: Example UI Pattern from http://ui-patterns.com/patterns/ProgressiveDisclosure

In terms of content output, tables allow users to see data in a well-structured, easily understood manner. Modifications may include the the presence of alternating row colors to allow for easier differentiation of lines, the ability to sort the table by any column, and filter possibilities. Furthermore, if the information available for each item in the table is too numerous to show in table format, some kind of progressive disclosure may be desired, in the form of accordions (for example, by allowing users to click on rows in a table to expand or collapse additional information displays). See Figure 2.4 for an example.

2.3.2.2 Layouting Web Applications

Separate from input and output concepts are layouting principles that allow developers to easily define the visual appearance of a site. Of particular note are the ”12 Column Layout” introduced by Bootstrap along with its grid and flex layouts.

Numerous libraries and resources exist to aid developers in designing frontend applications, including Google’s Material Design design language and Facebook’s Bootstrap CSS framework, the latter of which was used for the majority of the frontend’s development, with significant additions to styling and functionality.

2.4 Application Deployment

In any project with multiple developers or multiple devices, it is necessary to manage dependencies well to allow for frictionless development. The same is true for frontend development, testing, and eventual deployment. Due to the large variety of languages, frameworks, compilers and tools available to programmers, the exchange of package dependencies (with version numbers) is not a viable approach for maintaining portability in developing, testing and launching modern applications. The term container (“a box used for transporting goods so that they can be loaded easily onto ships and lorries”) [6] has been adopted by the software industry to describe the packaging of software in such a way that it can be *shipped to* (executed on) a variety of consumers without additional preparation. Software containers are lightweight, meaning they don’t include a full operating system together with the compiled software to be run and all necessary dependencies. Instead, only the source code and a reference to other container-based dependencies is included (along with certain configuration files and installation instructions). Multiple containers run on a single system with a shared operating system, and a shared container management engine. This lightweight approach makes it possible to host a large amount of containers on a single machine, as well as making cloud-based deployment more feasible (and faster). [12] [26] offers an overview of terms such as image, layer and container.

2.4.1 Docker

Docker is one popular container management tool, and includes tools to organize, package and execute containers and images. Docker has been likened to “chroot on steroids”, referring to a Linux command that changes the root directory of running processes, thereby making it impossible for the process to access files and commands outside of its directory [13]. The company Docker maintains the open-source software by the same name, originally released in 2013.

The architecture of Docker follows the server-client design pattern, with the Docker daemon acting as the host, and clients (such as the Docker for Windows client) accessing it via socket connections. The daemon manages the Docker containers and handles all client requests.

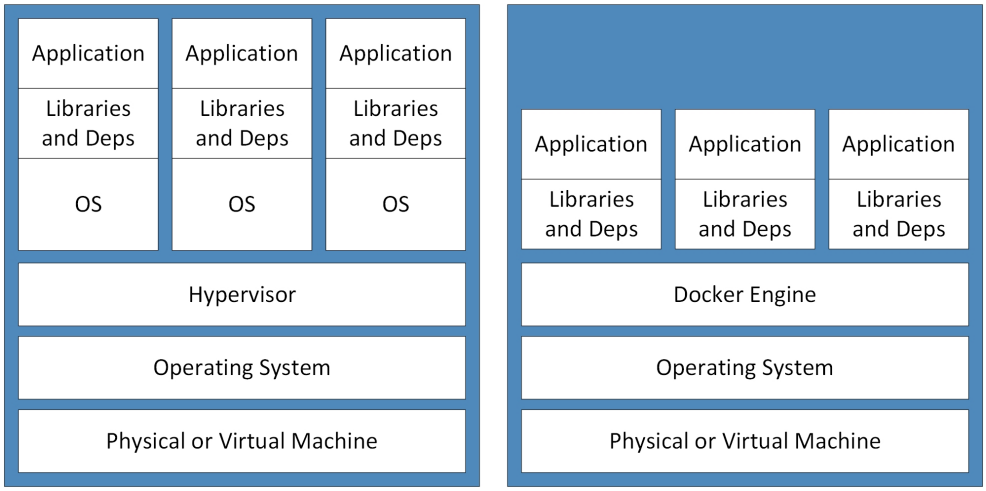

Figure 2.5: Traditional VM-based systems (left) vs. virtualized environments (right)

Several concepts surround the interactions possible with Docker, such as images, which are a hierarchical collections of files containing metadata on how to run a container, and containers, which are images that have been initiated and are running as a process. Docker images are stored in a registry. Figure 2.5 outlines how Docker-based deployments compare to traditional Virtual Machine - based server setups. The primary difference is that Docker containers all share a common operating system kernel, thereby greatly reducing the amount of allocated memory of any running services. [13]

Dockerfile

Images are built from so-called Dockerfiles, which contain dependency, configuration and execution instructions. A specialty of images created using Dockerfiles is the layer-based caching which occurs for every line. Specifically, each additional line in a Dockerfile creates a new immutable image based on the image created from the previous line's instruction. Instructions within the Dockerfile include commands such as FROM (to select a root image such as a Linux OS distribution, a ready-to-run database container, or even "SCRATCH", which is an empty image). Other commands include RUN to carry out installation or setup commands, ENV to define environment variables, COPY to include files and folders as part of the image, and CMD to start the service using a single command. [13]

Docker Compose

Using Docker Compose, multiple services can be bundled together using a single configuration file. This configuration file can also be overridden, allowing developers to execute containers for a variety of environments (dev, testing, production) with little to no changes to the image.

2.4.2 Alternatives to Docker

Docker isn't the only way to deploy software, of course. In [15], Krubner argues that Docker's advantages of security, portability and configurability aren't worth the complexity tradeoffs, and that developers lose control over their servers' infrastructure. He argues that writing simple bash scripts and setting strict requirements on servers' operating systems can save developers a lot of effort in dealing with the intricacies in Docker's cleanup of old images. Another alternative to Docker is Puppet, a tool to deploy software declaratively rather than imperatively. A so-called "PuppetMaster" calculates the minimal changes needed to achieve a specified system state and executes the necessary actions on all defined systems, although it only offers support for six linux variants including Ubuntu and SuSE. [3]

Chapter 3

Task Description

This chapter outlines the concrete requirements of the system, outlining the current state of processing reports by TWA employees, the specific use cases intended for the project, and a description of the intended user.

3.1 Current State

3.1.1 History of the Project

Before working on an internal solution, the TWA consulted with IBM to create a customized prototype based on IBM Watson, a proprietary AI-based analytics platform. Due to secure data storage concerns and other reasons, this approach was not sustainable. Instead, the system was implemented internally, in the form of a master student working alongside of a data scientist to produce a proof of concept system featuring basic text extraction, similarity matching, and result presentation capabilities. This thesis encompasses the continued internal development of this prototype TWADA application, aiming to further evaluate the feasibility of the system as an internal tool by.

3.1.2 How the TWA processes orders

When a client requires a component to undergo an inspection, he or she enters a work order into the TWA's order management system (with a short description of the kind of analysis that needs to be conducted) and selects a laboratory technician to carry out the order. When the assigned technician accepts the order, he or she can start analyzing the component, or optionally delegate subtasks (analyses) to other technicians. Further analyses are required frequently, as some inspections simply don't produce a clear result. The amount of effort that a given component takes to analyze often depends on which technician was assigned to the order – if an order is

assigned imperfectly, the corresponding component might take several days to correctly diagnose, costing valuable manpower and equipment time in the process. When an analysis for a given component/order is complete, the assigned technician authors a report outlining any and all findings. This report includes the original order description, the results obtained by analyses (including pictures), and a conclusion/recommendation. Technicians can also search the order management system for existing orders and reports with a matching title or description, however the non-standardized order entry form has few requirements regarding the level of detail in an order's description. Therefore, finding a match for a given order can be a challenge.

3.2 Target State

Figure 3.1 shows an overview of the intended use cases of the TWADA system. The identity of the main actor is further detailed in Subsection 3.3.

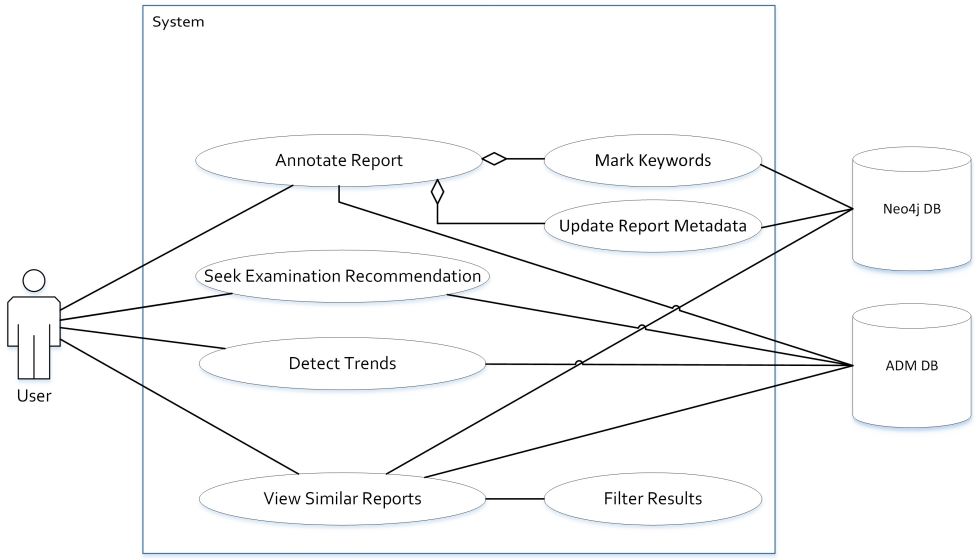

Figure 3.1: Use Case Diagram

3.2.1 Critical Features

Name	**Find Similar Reports**
Actors	User, (external ADM Database?)
Preconditions	The external ADM database has a record of the order, the Similarity Matrix (Model) has been trained, the server has imported the model and is ready for connections, and the user has accessed the website
Postconditions	The user can browse the list of search results
Description	The user enters the ID of an order into the textbox and presses the ”Find Similar” button. Next, the application displays a loading icon and communicates with the backend. Finally, the loading icon is hidden and the user is taken to the search results page, where a list of reports is displayed, sorted by similarity. If the server encounters any kind of error in the processing the request, an error message is shown to the user instead.

There is one primary use case that the system was designed to accomplish: Letting a user search for lab reports published to the TWA's internal server that match the subject matter of a given database entry. This use case has been split into two variants: The first variant uses a published lab report as the given database entry, while the second uses a newly accepted work order.

While finding reports that match the description of an incoming order might be a more helpful match for an analyst's workflow (an analyst has no need to compare completed reports with one another), the comparison of reports to reports carries less confusion (in terms of which parts of the report to compare) and was used as the basis for evaluating different versions of the system. The evaluation of the report can be seen in Section 4.3.2.4.

3.2.2 Desired Features

Name	**Annotate Reports**
Actors	User
Preconditions	The external ADM database has a record of the report, the server is ready for connections, and the user has accessed the website.
Postconditions	The user can keep annotating the report or go back to the start page.
Description	After selecting a report to annotate, the user navigates to the annotation editor. There, the text of the report is displayed, along with report metadata. The application automatically marks words that are already known. The user highlights parts of the report text and selects an annotation category, repeating this process any number of times. All selections remain marked for the remainder of the annotation session, showing their annotated category. Additionally, the user can edit some of the report metadata, such as its associated component(s) and COC(s). Finally, the user can press ”Submit” to save the annotations and metadata changes.

A secondary use case of the system is adding additional metadata to reports in the TWA's

database, and to actively improve the similarity detection capabilities of the system by annotating words and phrases contained within reports.

3.2.3 Optional Features

Name	**Detect Trends**
Actors	User
Preconditions	The knowledge base has been populated with several damage type annotations, the server has analyzed every report, the server is ready for connections, and the user has accessed the website.
Postconditions	The user is on the front page.
Description	The user looks at the front page's trends section, which shows a list of trend entries showing the amount, type and plant location of abnormally frequent damage types for a specific time frame. The user clicks on a trend entry to expand and again to collapse a detail view showing a graph of damage term frequency over time.

Name	**Seek Procedure Recommendations**
Actors	User
Preconditions	The external ADM database has a record of the order, the server is ready for connections, and the user has accessed the website.
Postconditions	The user is on the front page.
Description	The user enters the ID of an order and the application shows a preview of the order. The user clicks on the preview of the order to expand a detail view showing a recommended examination tool such as ”Rasterelektronenmikroskop (REM)” with a brief explanation as to why (e.g. ”The order deals with the component X and damage type Y. In Z percent of these work orders, this was the last and probably one of the most pivotal examinations”).

3.2.4 Nonfunctional Requirements

3.2.4.1 Adherence to Corporate UI Guidelines

At an early stage in development, it was decided that adherence to UI guidelines would be a significantly lower priority than the other functional parts of the software as a whole. These corporate UI recommendations include guidelines on color choice, layouting, usage of headers and footers, component usage, button design, navigation options, and font choice.

3.3 Personas

The application is intended to be used by only one group of persons, namely lab-based analysts of the TWA working on fulfilling incoming work orders. In terms of education, they may have received a college education with materials processing as their main field of study, however some kind of engineering background is expected. Alternatively, they may have received the bulk of their training as part of an apprenticeship inside of the TWA. Their experience can range from a few weeks to several decades of working inside the TWA, possibly resulting in a large discrepancy of work speed, pattern recognition abilities, and general domain knowledge among colleagues. TWA employees are also assigned to specific roles for each work order they are responsible for. When a work order is created by an client and accepted by a TWA employee, one analyst receives the role of ”Gesamtauftragsnehmer” (Overall order handler), who is then responsible for carrying out the order to completion. This overall order handler can then delegate individual examinations (”Teilaufträge” or partial orders) to ”Teilauftragsnehmer” (partial order handler), who carry out the specified analysis and report back their results. Because of the ”big picture view” of the overall order handlers, these are the intended users of the system. Using knowledge gained from using TWADA, they can hopefully minimize the amount of examinations (and staff) necessary to carry out an order. Theoretically, the system could also be used by clients of the TWA to be able to correctly assign their particular order to the right COC based on previous reports, however their inability to access the TWA’s ADM client effectively prohibits them from doing so.

Chapter 4

Implementation

This chapter explains the details of the implemented system, split into a system overview and several sections about the development of the frontend, development of the backend, architectural makeup of the knowledge base, and deployment of the application. Details of the frontend include an explanation of the organizational structure, behavioral logic, UI design, usability concerns and framework decisions, while the most important details of the backend include the general server architecture, third party library dependencies and how reports are processed, evaluated, stored, retrieved and annotated. Next, the structure of the knowledge base is illustrated in terms of types and relationships, and expected expansions. Finally, the deployment configuration is presented, detailing decisions about image choice, environment variables, caching priority, security, and external file system access.

4.1 System Overview

Figure 4.1 shows a bird's eye view of the entire system. A web application communicates with a web server hosting the application's API. This server interacts with the file system of the local machine and several external resources. These resources include an external Oracle database, an external file system (the department's DFS), and an instance of a knowledge base database running on the same machine. The similarity model found within the host system was to be generated via an external script.

Allocation of Work
The system was not developed by a single individual, but by a student working in conjunction with a data scientist from another department. In terms of responsibilities, the development of the web application, the setup of the server infrastructure, and the layout of the knowledge base were the responsibilities of the author of this thesis, while the analysis of reports, generation of the similarity matrix model, and original similarity finding algorithm were the responsibility of the data scientist. The backend server was a shared responsibility.

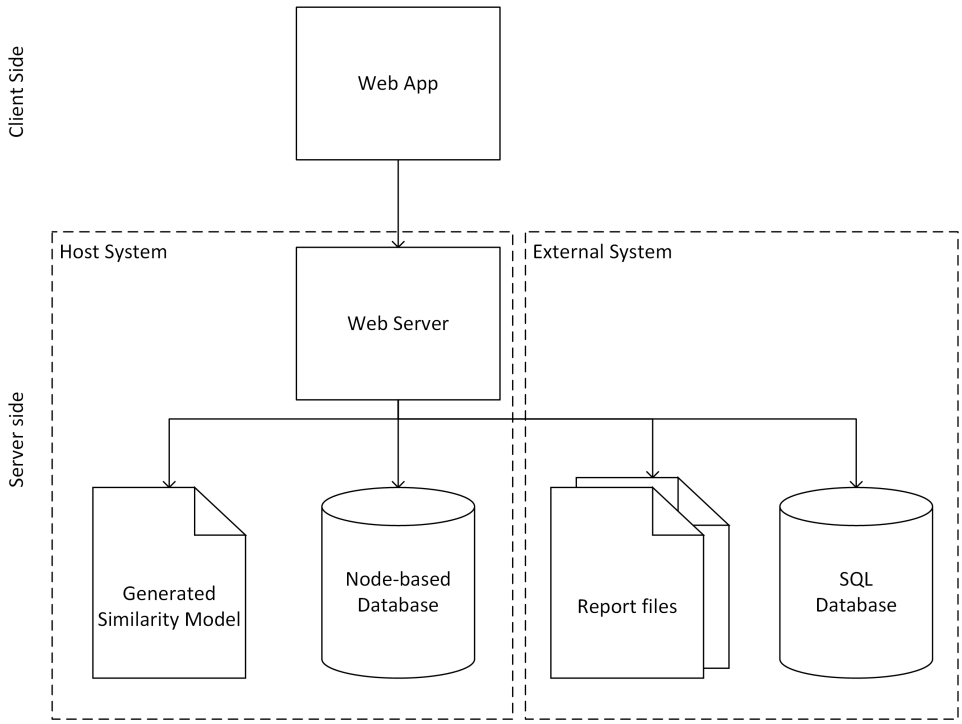

Figure 4.1: Basic System Design

4.2 Frontend

This section describes the main components of the web application.

4.2.1 Overview

The frontend of the app was created with the React web application framework, featuring a hierarchical composition of components. The original specification of the system foresaw the

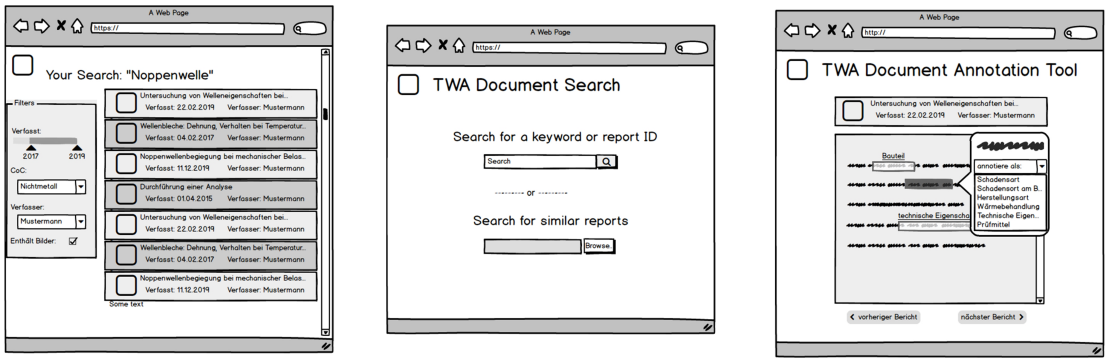

Figure 4.2: Three Views (Search Output, Search Input, Annotating

ability to provide a list of similar reports as a primary function and the ability to annotate text contents as a secondary function of the system. The system was therefore designed to facilitate three primary views within the web application, as seen in Figure 4.2 (from left to right): a view for showing search results (and being able to filter them), a view for inputting a search query (the default landing page), and a view for annotating reports.

The following sections explain the key implementation concepts of the main components, partially explained using the five separate concepts of the LDVCS paradigm (see [14]) and sorted by hierarchy (view, controller, logic, service, data). Note that the nature of a React web application makes a clear separation of these modules impossible, as every React component can exist independently while incorporating two or more of these concerns.

4.2.2 Search Page

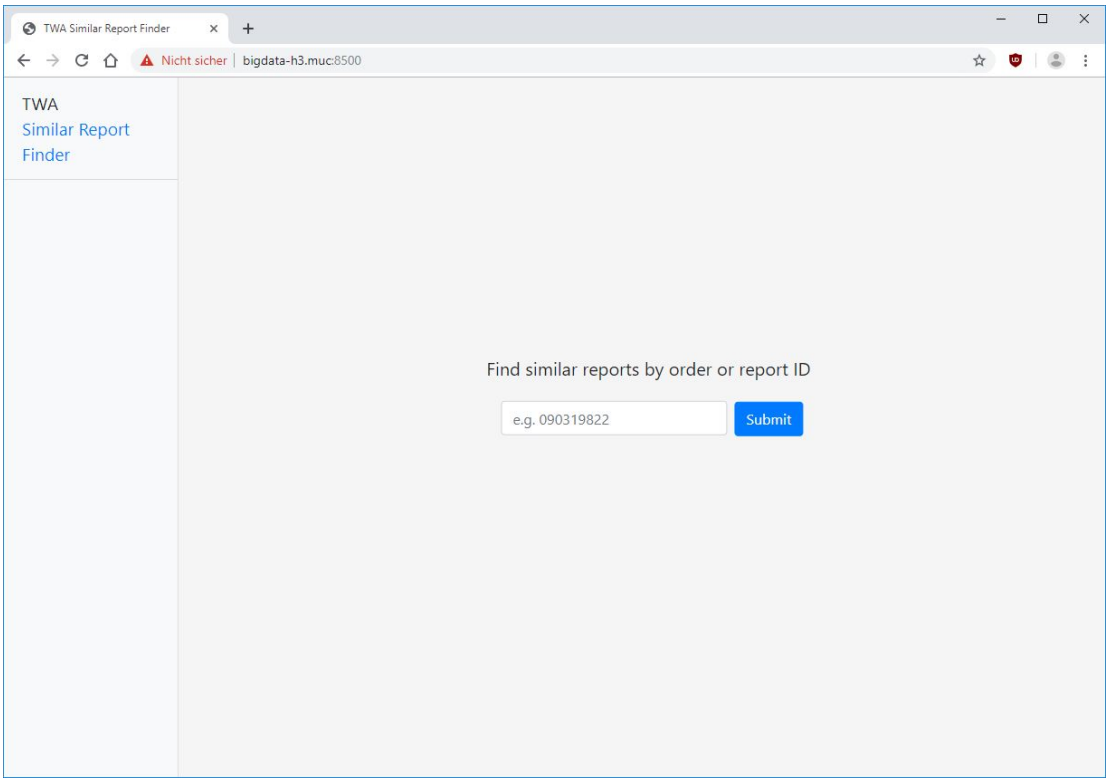

Figure 4.3: The Application's Start Page (Newly Loaded)

The landing page is the first view that users see when visting the website, allowing users to enter the ID of an order or a report and see a preview of the search object and optionally carry out a search or annotation action. Certain use cases also foresee this landing page being used to show workpiece analysis recommendations and detections of trends.

4.2.2.1 View

The front page of the application shows only an input field and a submit button at the beginning, and optionally renders ReportDetails and OrderDetails components as explained in 4.2.2.3.

UI Development
It bears mentioning how the final user interface was developed, as the landing page went through a number of design iterations in order to meet the requirements of all stakeholders in terms of usability and functionality.

Figure 4.5 shows the original design of the application, a Python application created with wxPython, a "cross-platform GUI toolkit" [9]. This less-than-portable desktop application took two inputs - a trained similarity model file and a Word or PDF document containing a report.

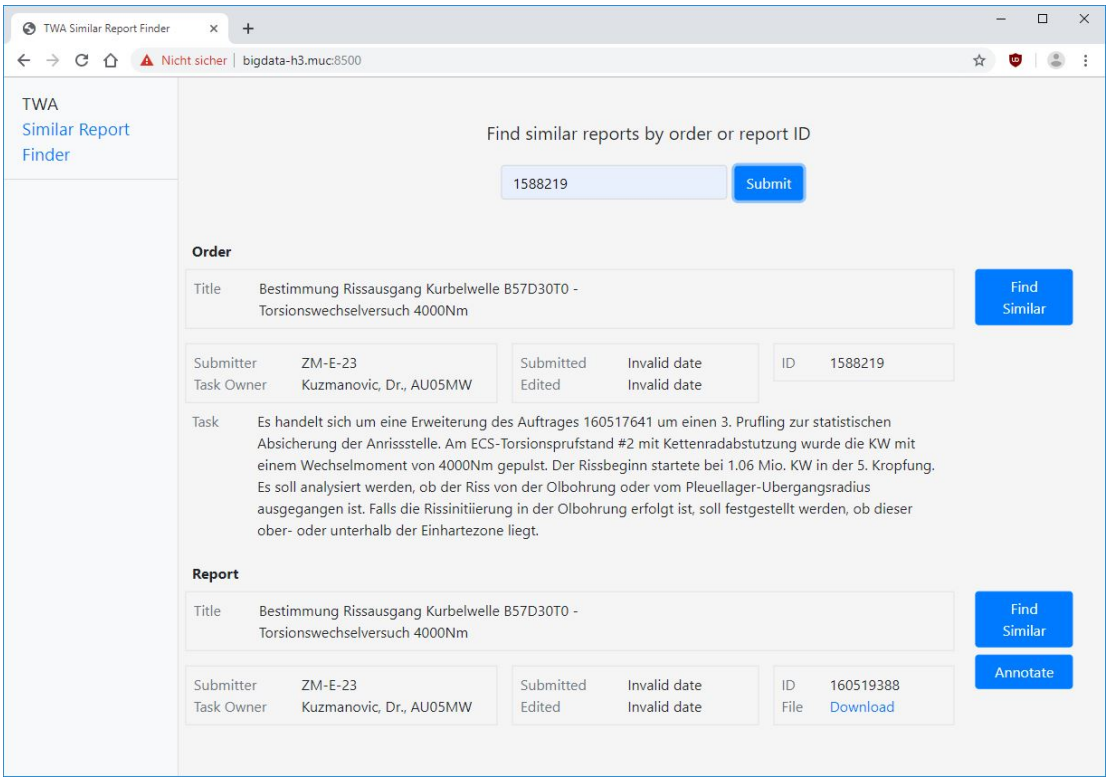

Figure 4.4: The Application's Start Page (After Fetching Information)

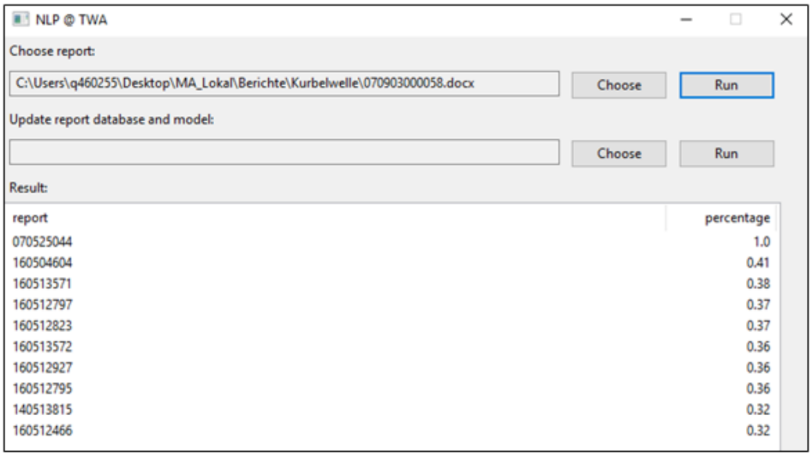

Figure 4.5: Original Design

At an early stage of development, mockups for the frontend focused on the theorized needs of the user rather than the realities of the backend's model. As seen in Figure 4.6, the design foresaw letting users search for topic keywords or upload reports to find similar ones, ignoring the use case of starting out with just an order.

Later prototypes such as Figure 4.7 prioritized a reduction of information redundancy, showing

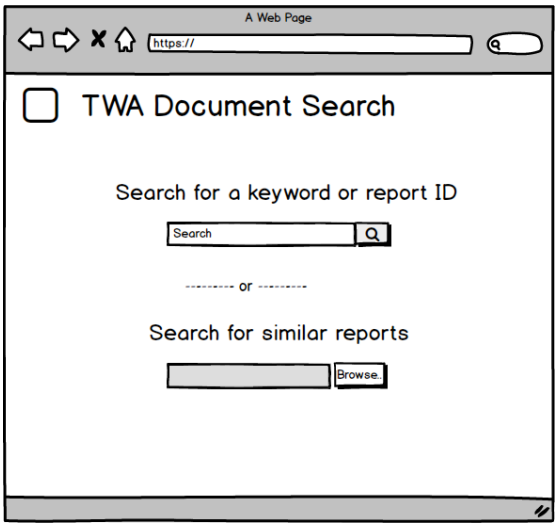

Figure 4.6: Early Design with File Upload

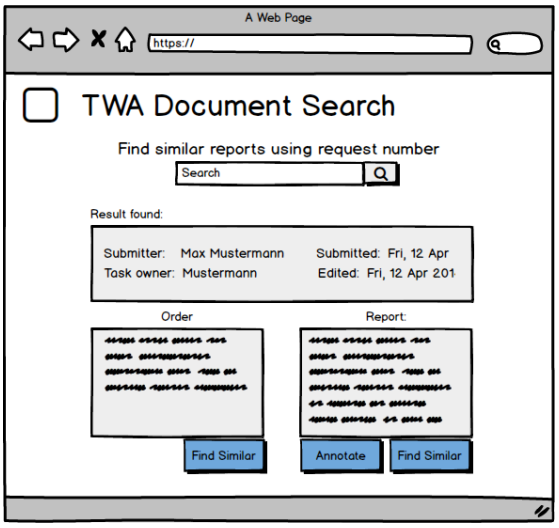

Figure 4.7: Later Design with ID Input

only a single field for metadata, then showing the reports individually. The redundancy of the current design was chosen to be able to reuse the ReportDetails component without a lot of changes.

4.2.2.2 Controller

In terms of acting on input, the front page (the SearchFormContainer) handles three input events:

Event	Triggered Action
Submit (ID)	Determine search object type and fetch metadata
Click ”Find Similar”	Fetch list of results from backend
Click ”Annotate”	Load Annotation Editor

4.2.2.3 Logic

The search component’s main responsibility is fetching report and order metadata for a given identification entered by the user.

The logic of the landing page is rather simple: The SearchFormContainer first determines whether the ID entered by the user belongs to a report or to an order, and then requests the corresponding object. If the API successfully returns a report, the application will then try to look for the corresponding order, and vice-versa (Note that some work orders don’t have an associated report). Two ”loading_state” variables are used to show this process in the ReportDetails and OrderDetails subcomponents. The subcomponents either show their current status (empty, fetching, none found), or show the details of the corresponding search object.

```
      //depending on ID format, searchObject has value "report"
      // and oppositeSearchObject has value "order"
      fetch(searchObject, id)
      .then((result) => {
        this.setState({ // update our component state to the returned data
          [searchObject]: {...result[searchObject], loading_state:"loaded"},
          [oppositeSearchObject]: {loading_state: "loading"},
        });
        return fetch(oppositeSearchObject, result[searchObject][
            oppositeIDfield]);
      })
      .then((result) => {
        this.setState({
          [oppositeSearchObject]: {
            ...result[oppositeSearchObject],
            loading_state:"loaded"
          },
        });
      })
```

Listing 4.1: Find Report, Then Order (Or Vice Versa)

4.2.2.4 Services

The front page component only connects directly to the web server. See section 4.3 regarding connected services and file systems.

4.2.2.5 Data

The data managed by the search page is limited to the user's input (the ID), and metadata of the report and order, including the title, department that submitted the order, the name and department of the person that handled the order and prepared the report, and the date of submission. The component also contains the work order's original description text to help users double-check their input ID.

4.2.3 Search Results

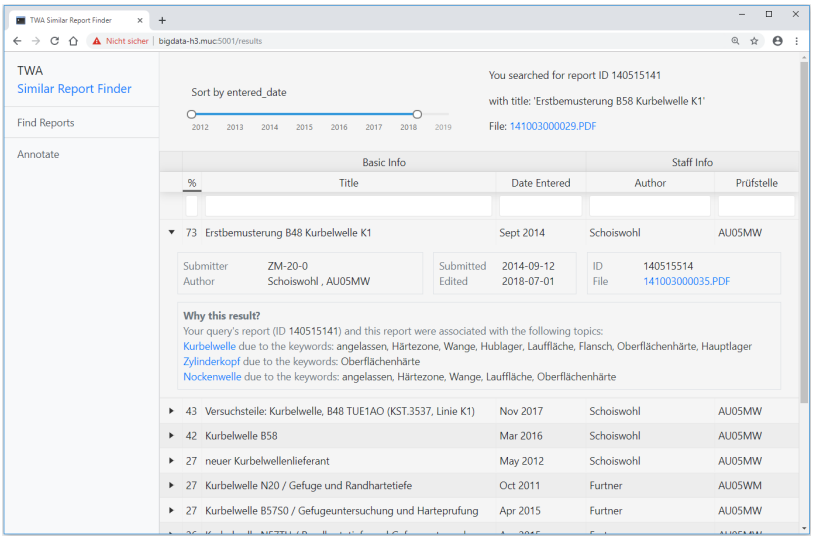

Figure 4.8: The Search Results Page

As its name suggests, the Search Results page (depicted in Figure 4.8) shows a table of related reports returned by the server. The user is navigated to this page when a "Find Similar" request (from any page) successfully returns. The table can be filtered and sorted by a number of categories, and entries can be expanded to show additional details or to carry out an action.

4.2.3.1 View

The final visual design of the page contains a remnant of the original navigation sidebar on the left, a date filter and a query reminder on the top, and a result table in the middle. Clicking on results inside of the table reveals a box with a ReportDetails component showing additional details. The date filter component uses Benjy Cui's rc-slider library [8], which allows the configuration of a multi-head slider with a custom range, tooltip, marks and notches.

The table is created using Tanner Linsley's React-Table library [19], which includes a large variety of configuration options for tables, the most important of which being the customizable expanding/collapsing container.

4.2.3.2 UI Development

During the implementation process, a number of additional UI variants were considered, two of which will be outlined here.

Per-Result feedback

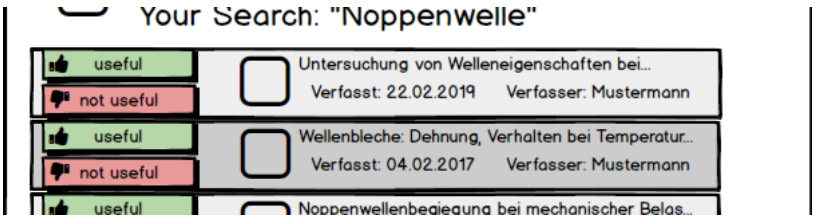

Figure 4.9: Conceptualized Feedback for Results

In order to help improve the model over time, we considered a way for users to give feedback to the system about individual results' relevance to the user's query. Figure 4.9 shows this. However, we lacked a way to incorporate this feedback into the model. Additionally, there was no way to indicate why a particular result was especially relevant or irrelevant for a particular input report, making it difficult to use the feedback for other searches.

Filtering

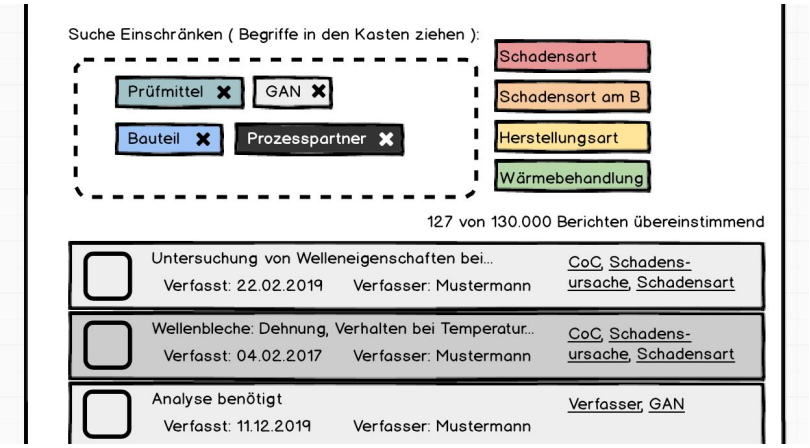

Figure 4.10: Conceptualized Click and Drag Filtering

Excluding the date, the only way for users to filter the rows of the table is by entering text in the tables' filter boxes (directly underneath the headers). This requires all filterable properties to exist as a header of the table. However, so many headings would either not be visible or make the table wider than the page, damaging usability. A custom filter box as seen in Figure 4.10 would let users restrict result reports to those matching the properties of the original query that they have dragged into the box. This approach was problematic due to the fact that many of the additional properties seen in the mockup (damage type, damage location, heat conditioning) aren't available as metadata, and would be dependent on the manual tagging process of reports, or an unimplemented / unreliable automatic detection.

Additional Variants
Other ideas considered for the Search Results page included thumbnail previews of the report's filetype, alternate pagination options (such as an infinite scroll), lazy-loading search results to avoid long idle times for users, and showing the textual contents of the report instead of the report's metadata.

4.2.3.3 Filtering Search Results

While filtering of the search results via the table's headers is handled by the third party table library, filtering the search results by the date filter is handled manually and serves as an opportunity to demonstrate how properties and functions are handed down in React.

Code listing 4.2 outlines roughly how the date filter modifies the visible results (see Figure 7.3 on page 49 for the component structure).

The following simplified code listing shows a portion of the Results component, the container of the Search Results page. As seen in the render() function, a FilterContainer component and a ReactTable component are initialized when this container is visible. The ReactTable component gets passed a data variable, calculated by taking the immutable props.rawData object passed to the container and applying a filter function (in this case, dateAllowed, which limits result rows to those within the confines of the state.dateFilter's start and end properties). The FilterContainer is passed a callback function (updateDateFilter()) that is called when the user drags the time range slider. This callback initializes or updates the value of the Results container's state.dateFilter attribute. When the state variable is updated, React notices the change and updates all components that are affected. In the case of the date filter, the ReactTable component would be rerendered, as it is being passed a value (data) that changes when the confines of the date filter change (via the dateAllowed filter method).

```
1  // From ResultTable.jsx (simplified and modified):
2  class Results extends Component {
3    //...
4    updateDateFilter(range) { ... }
5    dateAllowed(row) {
6      if (this.state.dateFilter) {
7        return (this.state.dateFilter.start <= row['entered_date'] &
8          row['entered_date'] <= this.state.dateFilter.end)
9      } else { return row; } }
10   render() {
11     return (
12       <FilterContainer onChange={this.updateDateFilter} />
13       <ReactTable data={this.props.rawData.filter(this.dateAllowed)}/>
14     ); }
15 }
```

Listing 4.2: Filtering By Date

4.2.4 Annotation Editor

4.2.4.1 Brief explanation

The purpose of the Annotation Editor is to allow lab technicians to enrich existing reports with manual annotations. These would improve the system's ability to classify them for the sake of providing more relevant search results for any given query.

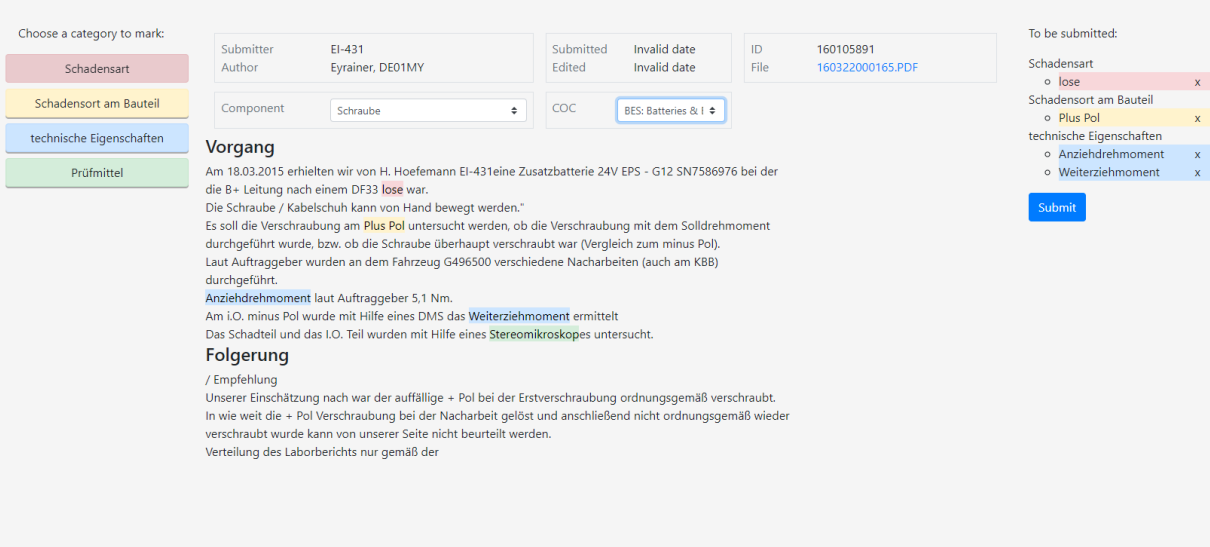

Figure 4.11: The Annotation Editor

Figure 4.11 shows the Annotation Editor, comprised of multiple subcomponents. On the left is a menu to select the current category that is to be marked. At the top of the editor is a ReportDetails component, underneath which lies the MetadataAnnotation component with two dropdowns. The two headings "Vorgang" and "Folgerung" denote the two SectionAnnotators components for each of the report's two sections. To the right of the editor is a the SubmittableAnnotations component, a live-updating list of annotation changes with the ability to synchronize these with the server when the user clicks the "Submit" button.

4.2.4.2 Data

The Annotation Editor is used to collect three pieces of data from each report: A list of car components that appear as a topic of the report being annotated, a list of COCs that were part of the report's investigation, and finally any new words that were marked by the user.

4.2.4.3 View

Category Selection
In order to collect this data in an efficient, user-friendly manner, certain design choices were

made. The category selection at the left was limited to to the four categories ”damage type”, ”damage location”, ”technical property” and ”examination method” in order to reduce the number of words that users would feel obligated to annotate and streamline the annotation process for a single report. The motivation for these efficiency considerations was the project's lack of information about the components that were investigated by the department. The only way to add the missing components to the system would be to ask lab technicians to annotate at least one report for each component, costing valuable manpower.

Metadata Multi-Select

In order to make picking components and COCs as simple as possible, the metadata is entered in the form of a multi-select dropdown form. For this, we made use of Jason Quense's react-widgets library, which allows users to add new options to a drop-down list if their intended selection is not available. The latter feature was especially useful for the adding of new components in an intuitive manner.

Existing Software

A number of existing annotation editors seen in Table 4.1 were evaluated in order to simplify development.

In the end, we settled on a minimal text-highlighting library to use as the basis of our editor. Martin Camacho's react-text-annotate library is a minimal module for highlighting text and selecting a category, outputting the annotated words in an easily processable Javascript object.

Use Case Walkthrough

The following describes how annotating a report happens, from the perspective of the user. When the user wants to annotate a particular report, he or she enters the ID of a report into the front page's search box, waits for the report to appear, and presses the ”Annotate” button, whereupon the user is navigated to the annotation editor view. Next, the annotation editor loads the report's text and metadata, and automatically highlights any words already known by the system. The user can now choose to either edit the report's metadata (limited to components or COCs), or to annotate parts of the report by selecting a category to highlight, and then marking some text. A list of the user's changes is displayed on the right side, with the ability to unmark individual words.

Name	URL	Comment
TagTog	http://docs.tagtog.net/	Propriety annotation editor software requiring a paid license.
Doccano	https://github.com/chakki-works/doccano	Django-based, Docker-supported library with questionable interoperability.
Webanno	https://webanno.github.io/webanno/downloads/	Java-based annotation plugin; can't be integrated into our web app.
NeoOnion	https://github.com/FUB-HCC/neonion	Django-based tool made for documents rather than text with arguably unintuitive UI.
Pundit 2	https://github.com/net7/pundit2	Outdated library with no support.
H	https://github.com/hypothesis/h	Propriety annotation editor client with free license, but the provider has access to all annotation data.
Annotator (lib)	https://github.com/openannotation/annotator	Outdated library with no support.
React-Annotation	https://www.npmjs.com/package/react-annotation	Image-based (not text-based) annotation software.

Table 4.1: List of Existing Annotation Software [Libraries]

4.3 Backend

The serverside parts of the system, as outlined in Figure 4.1 on page 22, contain a webserver (also referred to as ”the backend”) with access to local files, external files, and two database systems. This section outlines the structure and behavior of the server in terms of basic architecture, report processing, and request handling.

While its details will be explained here, the implementation of the backend system was initially not part of this thesis.

4.3.1 Overview

TWADA's backend consists of a single Python Flask-based server serving both the web application's static HTML files and handling all API requests, configured to run with multiple threads. In production, the server requires the TWA's file system to be mounted under a specific folder.

4.3.2 Primary Backend Functions

The backend system, in addition to serving the files of the web application, has a number of responsibilities:

- Supply information on orders and reports
- Offer reports for download
- Allow the addition of new modification of existing keyword matches
- Find similar reports (for orders and reports)

4.3.2.1 Supply Information on Orders and Reports

Basic information about orders and reports is necessary for every page of the web application. The data helps users verify they have entered the correct ID on the search page, be reminded about the basis for the current page of results, preview information about current search results, and double-check details about the report they may be annotating. This basic information includes the report's or order's title, author, date of submission (and last edit), ID, and the relevant department information. The backend gains access to this data via the OracleDB-based ADM ("Auftragdaten Management-System") database hosted by the TWA department.

4.3.2.2 Offer reports for download

The frontend's ReportDetails component includes a download link for the report. By being able to view a direct version of the report helps analysts use the results of the application quickly, as well as increase trust in the system by tying unfamiliar detail views to real documents that the user is accustomed to working with. In order to determine the path for a report with a given ID, the system queries the ADM database, and then accesses the file from the external WinDFS-based system. Earlier versions of the system used local file links to link to reports, with the help of an extension. Using these client-based network paths had the advantage of using existing network path permissions, however technical complications (the file paths didn't work for all users, possibly due to missing extensions, operating system incompatibilities, and differing network path mappings) and permissions issues (every user using the system would need to individually apply and be approved for folder permissions) encouraged a more standard, server-based file download.

4.3.2.3 Allow the addition of new modification of existing keyword matches

Using the annotation editor, users can add new keywords, components and reports to the system, as well as modify existing relationships between reports, components and COCs. As detailed in Section 4.4, all keywords used for the categorization of reports are stored in the knowledge base, therefore the realisation of this function requires no other systems.

4.3.2.4 Find similar reports (for orders and reports)

Finding similar reports is the most important function of the system, and will be explained here in detail. Figure 4.12 shows the network requests necessary to accomplish the task, assuming the server has loaded all other necessary information at a previous point in time.

Text Extraction
When the user clicks the "Find Similar" button in the frontend, a request is sent to the backend containing only the ID of the report or order in question. Figure 4.12 shows the network requests necessary for the request. The backend's first action after receiving the request is to determine the report's download location (from the ADM database) and then opening the file (from the external file system) to extract the text. In the case of finding similar reports for an order, no report file exists yet from which the text can be extracted. Instead, the system takes the title and description fields belonging to the order as saved in the ADM database and uses these as the basis for comparison. The text extraction step for reports makes use of several Python libraries to handle the processing of PDF and Word documents, namely Textract as a wrapper for pdftotext and antiword.

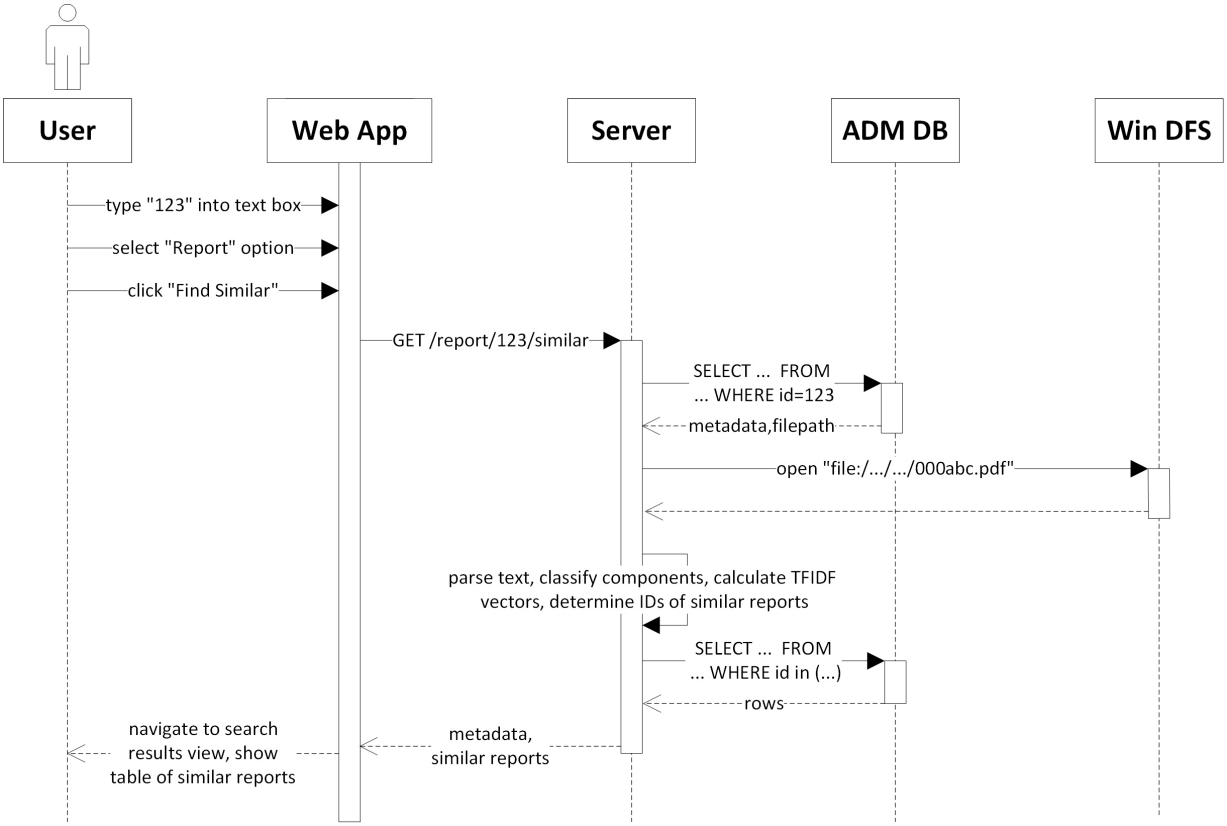

Figure 4.12: Sequence Diagram for ”Find Similar Reports”

Text Processing

After extracting the raw text, the backend tries to separate the text into sections by looking for headers of the expected text sections ”Vorgang” (procedure), ”Ergebnisse” (results), and ”Folgerung” (conclusion), although the latter is often called "Empfehlung" (recommendation"). The "Vorgang" is often equivalent with the ”description” field of the order, although this text may have been expanded upon by the report author in order to describe in more detail which analyses are being conducated. The ”Ergebnisse” field contains a summary of the report’s findings, such as an evaluation of a workpiece’s damage or a confirmation of the validity of another observation. The contents of the ”Vorgang” and ”Folgerung” fields are used as the basis of the similarity analysis. The ”Ergebnisse” section houses the quantitive and qualitative data gained from analyses. Due to the large variety of analyses and variations of analyses conducted by TWA employees, no set of standards for the formatting of these results exists. Indeed, the large variety of partially color-coded, partially labelled graphs, tables and oftentimes freeform text formats has made the automatic acquisition of knowledge from this section virtually impossible despite repeated customer requests for the inclusion of this data in a similarity analysis.

After the extraction of the report’s contents, the text undergoes a preprocessing step to filter out semantically unimportant contents. These include whitespace characters, universal industrial abbreviations such as DIN and ISO, special characters, numbers and common words in the form

of stop words (see [23]).

After removing superfluous words, all remaining words of the report are stemmed in an attempt to take the base form of words which have specific tenses and conjugations applied. While lemming these words would theoretically produce the true roots of the systems (see [20]) and German is a high-resource language, the nature of the report's highly domain specific technical phrases makes this approach unfeasible. At this point in the process, the report's TFIDF vector is calculated.

Document Classification
The next step in finding similar reports is determining the possible components that could be the subject of the report. This categorization step uses the stemmed version of the keywords found in the neo4j database, together with their component associations. Because the generated similarity matrix is neither re-generated when new orders and reports are added to ADM, nor when new components (and keyword mappings) are added to the knowledge base, the categorization can only take place using components known when the server starts, at least in the current version of the software. As a result of this restriction, the keywords for identification of components are not loaded from neo4j directly. Instead, we use the keywords imported from a manually created JSON document containing the results of a request to the server's /query endpoint, which returns the contents of the knowledge base in terms of keywords, stemmed keywords, and associated components.

As detailed in Section 4.3.2.4, our document retrieval system uses a multi-step process to identify reports that contain similar contents as a given search object. In the first step, the possible topics of the search object are determined. As part of this step, an input report's extracted and processed conclusion text could, for example, be classified as matching the topics "Kurbelwelle" and "Nockenwelle" due to the presence of characteristic keyword stems. Previous internal efforts of the TWA used machine learning techniques to identify these topics, however this input corpus was limited to reports with the topic name in its title, thereby potentially training a reliance on the report title and causing the model to be equivalent to a pattern-matching rule. The current implementation employs a simple keyword-based classification scheme, although an improved machine learning based approach could be used instead.

Similarity Identification After having identified the set of possible components that the analyzed report might have as a topic, the algorithm isolates only those reports in the similarity matrix that also match these topics. Next, the algorithm processes the TFIDF vectors separately for "Vorgang" and "Folgerung" within these topic groups (e.g. among reports that may be related to the topic "Kurbelwelle") to calculate the most similar matches. A specific similarity score weighing scheme is applied depending on the type of the search object (for an order, no "Folgerung" matches are possible), and to ensure that similarity scores are within a reasonable margin. After duplicate matches are discarded, the metadata for the results is requested from the ADM, and the results are returned to the web application.

Altogether, four kinds of resources are required to accomplish this task, these being the ADM database for basic information about the report (or order) in question, the knowledge base for

finding keywords that can classify the report, the generated similarity matrix for comparing TFIDF vectors, and the WinDFS file system for opening, extracting and parsing the text of reports.

4.3.3 Secondary Backend Procedures

In additional to the aforementioned primary tasks, the server also carries out a secondary task, namely the optional re-initializion of the graph database. A further secondary task, the generation of the similarity model, is carried out by an external script.

4.3.3.1 Excel to Neo4j Transformation

The initial source for keyword associations for the various component categories was a cooperatively edited Excel spreadsheet (Keywords.xlsx in the twa_app directory). In order to be able to reset the knowledge base to a reproduceable, non-empty state during development, a Python file was written to ingest the keywords (excel2cypher.py). The parser is also able parse the color information contained within the cells indicating the weight of the words. As part of this Excel file, cells are colored green, yellow, and red to to indicate the estimated correlation between a keyword and the category of the current sheet in question (For example, the term ”Zündkerzenbohrung” (spark plug drill hole)’ under the heading ”Schadensort am Bauteil” (damage location) is marked green to indicate that it can be used to unique identify a report dealing with a cylinder head). This human-applied color weighing is saved in the knowledge base as a ”weight” variable of the ”APPLIES_TO” relationship between keyword nodes and category nodes.

4.3.3.2 Similarity Matrix Generation

The generation of the similarity matrix is carried out by an external Python script which must be run manually. The script follows nearly the same procedure carried out in Subsection 4.3.2.4. Starting with a large list of file paths, the script extracts the text of every indexed German report present in the TWA’s archives, determines their possible topics using cached neo4j keywords, and calculates a large TFIDF-based similarity matrix. Much of this processing is done in parallel.

4.4 Knowledge Base

The implemented system uses an RDF database (specifically Neo4J) to store information necessary for the classification of reports. The database includes data on entities such keywords, components and centers of competence (COCs) as well as their assocations between one another.

4.4.1 Purpose

Actual: - Storing keywords - Storing components - Associating keywords with components - Storing references to reports - Associating reports with components

Potential: - store semantic information about reports - automatically derive properties of report from automatic annotation of reports -

4.4.2 Structure

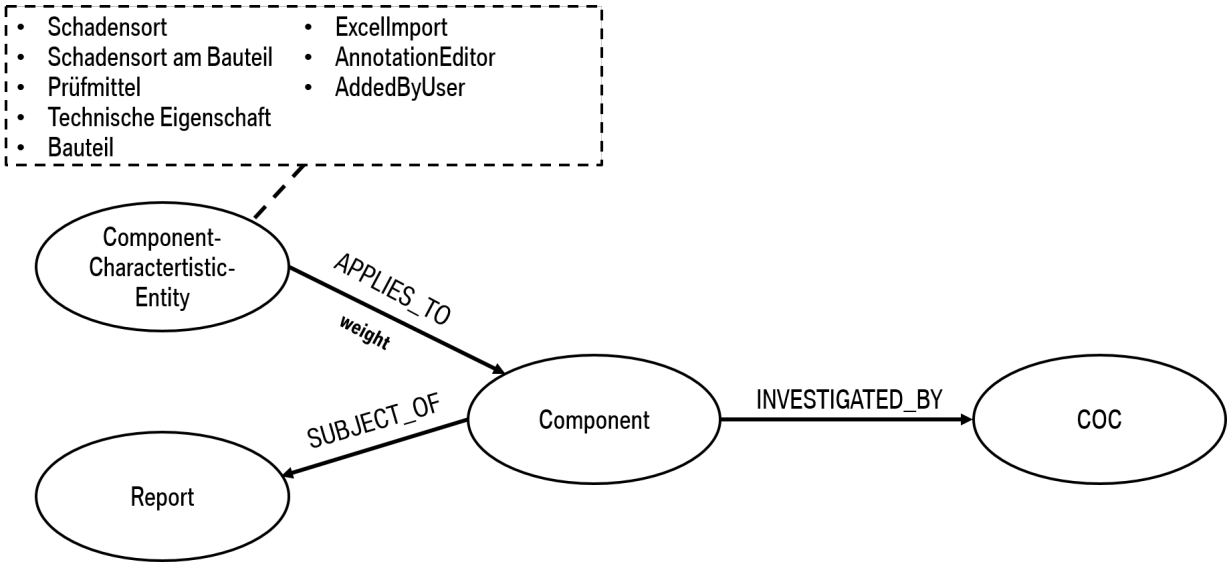

Figure 4.13: Ontology of the Knowledge Base'

Figure 4.13 shows the types and relationships present in the system's knowledge base. Keywords (called CharacteristicComponentEntities for legacy reasons) are associated with components via the APPLIES_TO relation mentioned in Section 4.3.3.1. Any number of keywords can be associated with any number of components, although components with no matching keywords will never be found during report classification. Components can be the SUBJECT_OF of any number of reports, and vice-versa. In an effort to aid the assignment of orders to COCs in the future, the INVESTIGATED_BY relation captures the corresponding COC of a component.

CharacteristicComponentEntities are the overarching class of all keywords, however multiple subcategories exist. The annotation editor allows users to pick some of these, including damage type ("Schadensart"), damage location ("Schadensort am Bauteil") and technical property ("technische Eigenschaft"). Other subcategories of keywords describe the source of their addition to the knowledge base, such as stemming from the initial Excel import (Section 4.3.3.1), or stemming from a user's usage of the annotation editor ("AddedByUser" and "AnnotationEditor"). Future expansions of the knowledge base may include additional information, such as the identity of the user making the annotations, or the date and time they were added.

4.4.3 Decision to Use a Knowledge Base

The decision to use a knowledge base was not made as part of the architectural design process of the project. Instead, it was already decided upon prior to the start of development, although it was erroneously referred to as an ontology. We hope that future development efforts will make even more use of the node-based database's structural benefits than TWADA does currently.

4.5 Docker Deployment

BMW's TWA department was not able to provide its own hosting equipment for running the webserver, and security guidelines prohibit the use of external hardware and services, thereby limiting options for deployment of the web application. As a fallback option, access to a shared interdepartmental development server hosted by BMW's Data Science department was sought and granted. The server's infrastructure prevented users from installing any programs (such as Python), but allowed the execution of Docker containers. The application was therefore deployed as a Docker container, enjoying the benefits of immutable builds and easier dependency management as a side bonus.

See the addendum for a fully annotated listing of the final Dockerfile. Compiling an image using the Dockerfile starting with no cached image can take between 30 and 60 minutes.

4.5.1 Deployment with Docker

This section outlines decisions, troubleshooting steps, and eccentricities encountered during the creation of a stable Dockerfile that can compile and run the TWADA application.

4.5.2 Base Image Selection

The first line of a Dockerfile (ignoring comments), and thus the first decision for developers of a new Docker image is the base image selection. Due to the many benefits of a terminal-first operating system and the relative lack of compatibility issues with many Python packages, we wanted to use some Linux-based image. One popular base image at the time was Alpine, a minimal Docker image costing only 5MB. However, the expected size of the final image due to all the packages that would be installed and the size of the code and models copied during the image build step (>1.5 GB) dissuaded us from this approach. We instead chose Ubuntu due to personal familiarity and the large amount of community support available. An alternative to raw Ubuntu would be using an existing "data science" image containing, for example, a full installation of Anaconda along with multiple data processing Python packages. However, some of our libraries and plugins require very specific versions of other libraries, making pre-configured images unappealing.

4.5.2.1 Environment Variables

Although there are seemingly multiple ways to pass environment variables to a docker process, not all of them work. Our initial attempt at passing variables via the standard Debian-based EXPORT command (e.g. ”RUN EXPORT myvar=myvalue”), for example, was problematic, as we failed to account for Docker’s layer-based images, wherein only some operating system states persist across layers. Changes to the file system carry across ”layer lines”, for example. Environment variables do not, however, unless set using Docker’s special ”ENV” command.

4.5.2.2 Caching Behavior

The final Dockerfile features nearly 70 steps. This number could be drastically reduced (for example, all software packages and libraries could be installed in a single line), however the high number of steps carries very few detriments. Instead, it tends to decrease the amount of time it takes to recompile an image for any change therein, due to the caching behavior of Docker. Roughly like an onion, Docker adds layers on top of previous layers. Any layer that was successfully built in a previous step will be saved to the build computer as its own image. Changing the version of a library, for example, or changing a line of code, will require less time in rebuilding the image if the majority of build steps take place before the line where the change takes place. This also explains the very exact copying of files and folders to the Docker image.

4.5.2.3 Passwords and Security

Passwords are passed to the Docker image via environment variables at runtime, using variables supplied to a Bash script run by an administrator. We use the Docker process’s owner (and supplied password) as one set of credentials used to connect to the internet via a corporate proxy. This set of credentials is used to download libraries and install packages. Once finished, we overwrite the proxy settings and use only the credentials of a technical user, who does not have permissions to access the internet but does have the ability to connect to the WinDFS file system (explained in the next section) to download reports.

BMW’s password policy causes periodic resets of users’ passwords. Unfortunately, the environment variables for the proxy are set at the beginning of the Dockerfile. Therefore, if the image needs to be recompiled for any reason, nearly the entire cached layers of the Dockerfile are invalidated.

4.5.2.4 Windows Distributed File System

The TWA uses Microsoft’s Distributed File System [18], allowing users to seamlessly navigate to network-based subfolders file system using their Windows file browser. As part of the deployment of the application, the server needed access to the file system containing the department’s

reports. It was discovered that Docker images, unlike Virtual Machines, share the kernel version of the Docker daemon running on the local machine, and do not feature 100% reproducible builds. In the case of a local Windows machine, the Docker for Windows daemon had a Linux kernel that did not support mounting CIFS (common internet file system) paths, even when updated to the latest version. Fortunately, the Docker daemon of the development server of the data science department had the necessary linux kernel version to be able to mount the file system.

4.5.3 Automated Deployment

BMW licenses the Openshift container platform, a SaaS solution to simplify the creation, hosting and management of container-based services, including options such as continuous integration and delivery. Standard cost accounting procedures would, however, necessitate having the TWA set aside financial resources in order to benefit from the licensed software. As the development of the TWADA prototype did not foresee a budget for software, this was not possible.

Instead, we experimented running bash scripts in parallel that would repeatedly poll the repository until a new commit was added, rebuild the image using the new code, and kill and restart the Docker process when the new image completed building. After the Docker process ended (for example, because it was ended by a parallel script), control would be returned back to the bash script, which would then start polling for changes again. Having two such scripts running in parallel was sufficient to emulate a continuous deployment server, automatically rebuilding and redeploying changes for any commit. However, unforeseeable errors during the build step without proper logging made problems hard to diagnose.

Chapter 5

Summary

John Tukey, father of exploratory data analysis, once stated ”the combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.” While our system is limited to the data (and the format of data) contained within the TWA's archives, we would argue that our system does provide users with a reasonable answer in a majority of cases and successfully fulfills all required functions. Using the implemented system, TWA employees can easily search find reports that are semantically similar to work orders and existing reports. Furthermore, TWA employees can use the application's annotation editor to improve the similarity detection by adding new components and identifying keywords via a user-friendly editor. The system has also been evaluated by five employees of the TWA so far, with overall positive feedback regarding the search results and usability of the application.

Altogether, the implemented prototype enables BMW's TWA department to save on manpower and equipment costs while reducing the time it takes to process orders. While further development is necessary to gain full functionality of the system, selected examples have already demonstrated the system's abilities.

The fields of natural language processing, knowledge management, application development and information technology operations are ever-expanding. TWADA represents a hallmark of complex software engineering, integrating multiple database technologies, data analysis methods, user interface concepts, and employing modern standards of server deployment.

Chapter 6

Future Works

While TWADA is a fully working prototype, areas for improvement do exist. The most important ones are presented here.

6.0.1 More Data

In general, a more in-depth evaluation of the prototype would best be conducted using a larger amount of data, either by having the department dedicate manpower to the annotation of reports, or by automating the process of finding identifying keywords. The latter would be possible given a list of unknown components, by having a script naively match the titles of reports with the component names in the title, and then marking the highest-TFIDF-scoring words as keywords that identify this component.

6.0.2 Automated Model Generation

Another area of interest would the automating the generation of the similarity model, ideally every time new reports are added to the ADM system. The TFIDF similarity matrix is currently generated manually using an external script, however this could be added as a scheduled or event-subscribed task to the system. Because the generation of the entire model takes a lot of resources (and because most TFIDF vectors wouldn't change much with the addition of a few reports), a partial model update may also be possible.

6.0.3 Extension of the Knowledge Graph

The nature of graph databases allows for easy extension and modification of existing data structures. The system could benefit from increased data connections between keyword nodes, by identifying synonyms or frequently present neighbors of words.

6.0.4 Extension of the Annotation Editor

Some additional features could benefit users interacting with the knowledge editor, such as the ability to individually assign keywords to components, or the ability to remove incorrect annotations that were previously marked and saved by other employees. Alternatively, all annotations could be submitted to an administrator for approval.

Keyword Association
One problem that also requires a solution is the fact that annotated words are only associated with the component that they were originally saved with. If enough keywords are present in the system, it is possible that an analyst using the annotation editor may encounter a report whose contents have already been annotated, but cannot be associated with the component of the current report. One alternative would be associating every known keyword with every component of every report that the keyword occurs in, when saved using the annotation editor. However, this may could a large amount of keywords to be associated with a majority of components. A more sophisticated classification algorithm could solve this issue.

Chapter 7

Addendum

This section details additional implementation details not contained within the body of the thesis, including an overview of the component hierarchy of the frontend, a short description of the system's API endpoints, a summary of the files and folders present in the backend, an overview of which modules are imported by which backend module, and a short guide to reproducing the developers' development environment as a crutch for future implementers.

7.1 Frontend Documentation

The React-based web application consists of a number of subcomponents which combine to form an interactive single page web application. When a browser navigates to the page, the server returns an HTML file. This file loads a script which instantiates the web application by attaching the aforementioned subcomponents to the site's DOM.

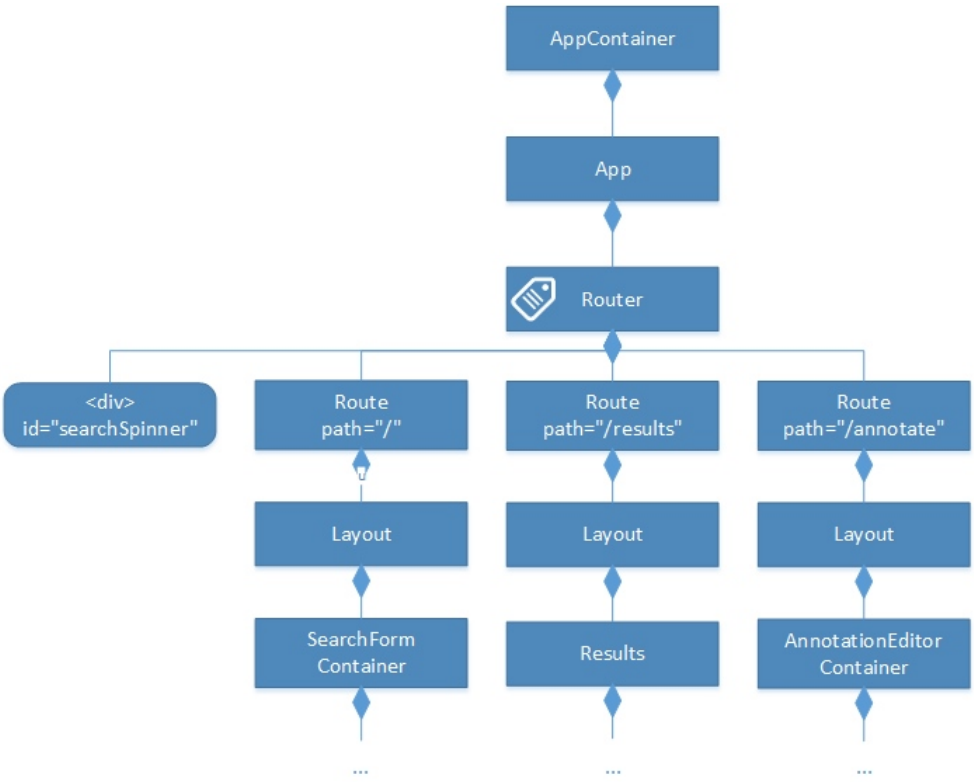

Figure 7.1: Components Top

7.1.1 Component Hierarchy

7.1.1.1 Root Container

The application makes heavy use of the Presentation / Container Component pattern, wherein the data (state) and logic of a single organizational unit are kept separate from the controller (event handlers) and view (HTML elements) to improve code maintainability and reduce coupling.

The root component of the application's component tree is AppContainer, housing functions that need to change the state of the entire application. Among these, "doSearch" and "annotateReport" navigate to the "/results" and "/annotate" pages respectively. AppContainer contains "App", which houses Route components from the external library React-Router. These Route components enable URL-based navigation within the single page web app. Inside each "Route", Layout components act as a wrapper, surrounding the child nodes with a uniform structure (this renders a sidebar to the left of each page). These child nodes (SearchForm[Container], Results and AnnotationEditor[Container]) are the three primary views of the web application.

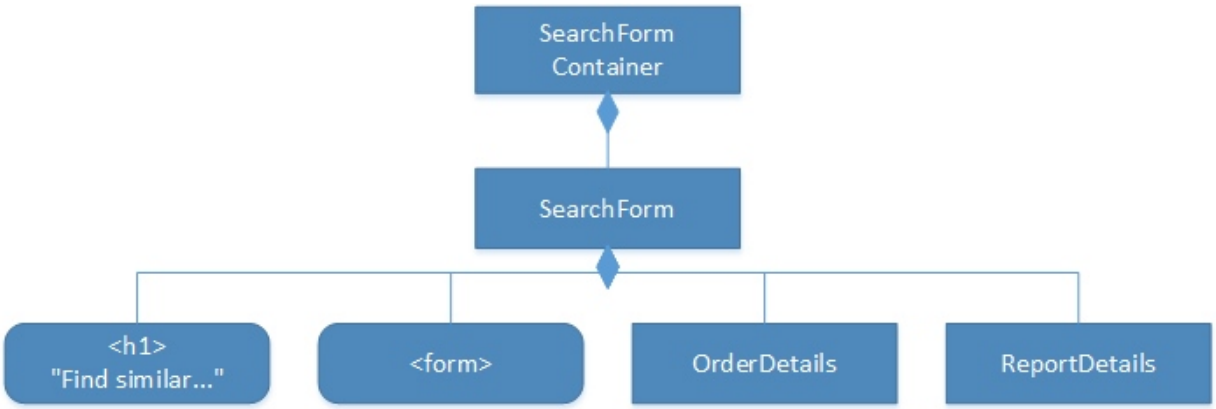

Figure 7.2: Search Component

7.1.1.2 SearchForm Component

The SearchForm is the default view of the web application, allowing users to input an ID, view report and order details, and navigate to the other views via those detail views. The component's container includes logic to handle input change and submission actions.

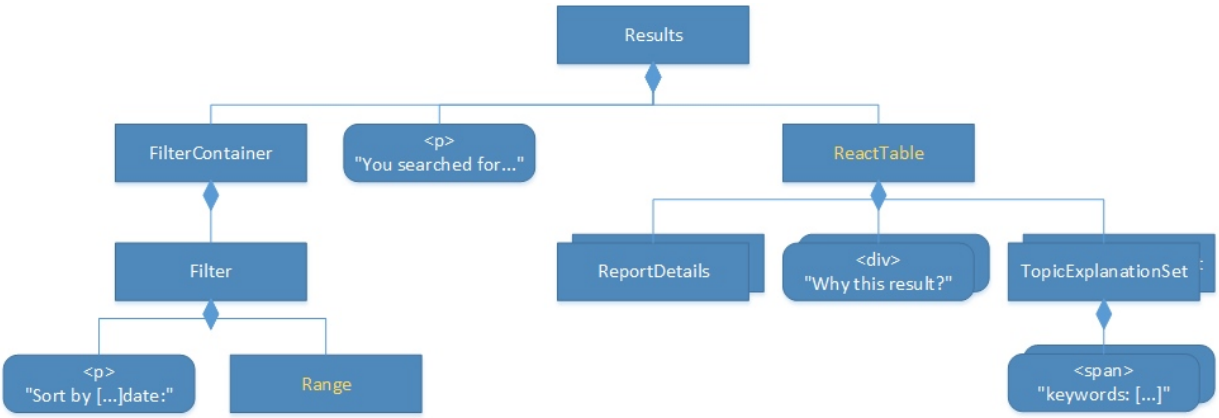

Figure 7.3: Results Component

7.1.1.3 Results Component

The Results component is responsible for displaying a list of similar reports for a given query. The component's children are FilterContainer, a showing the user the details of the item they just searched for, and a customized ReactTable component from the external "react-table" library. FilterContainer's Filter child includes a Range component from the external "rc-slider" library, featuring a slider with multiple handles that lets users filter the currently visible report results based on their creation date. The ReactTable component renders a customized table. Each row of the table features an expandable accordion component that renders a ReportDetails component and an explanation for why the report of the current row was deemed "similar".

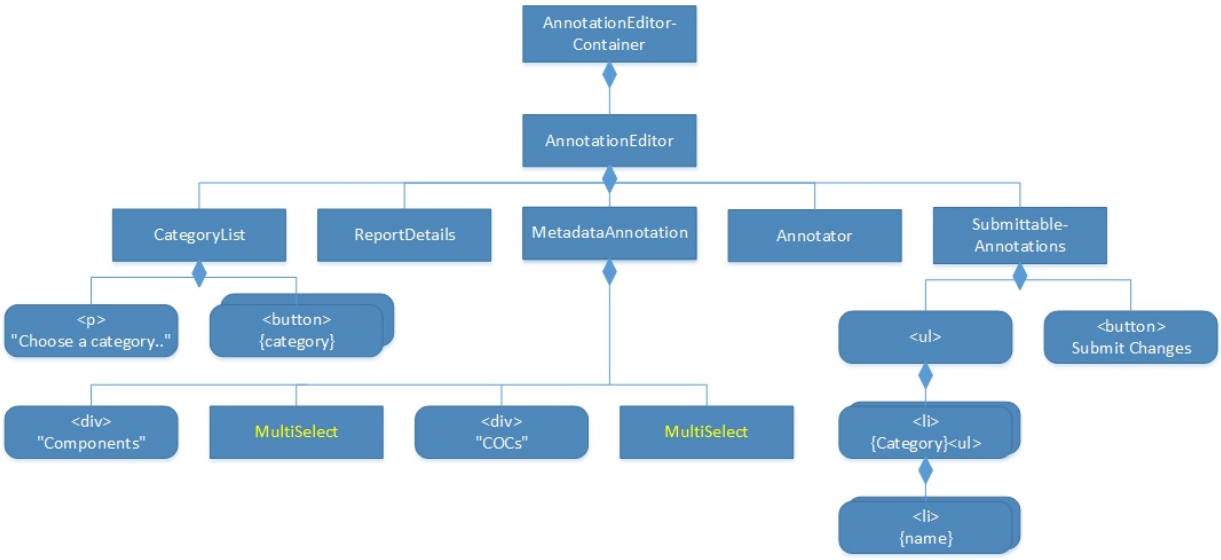

Figure 7.4: AnnotationEditor

7.1.1.4 Annotation Editor

Figure 7.4 shows the AnnotationEditor view's hierarchy of components. Contained herein is CategoryList, a column of buttons which users can press to switch the current highlight category; a ReportDetails component; MetadataAnnotation, a row of multiselect boxes from the external React Widgets library which lets users change the

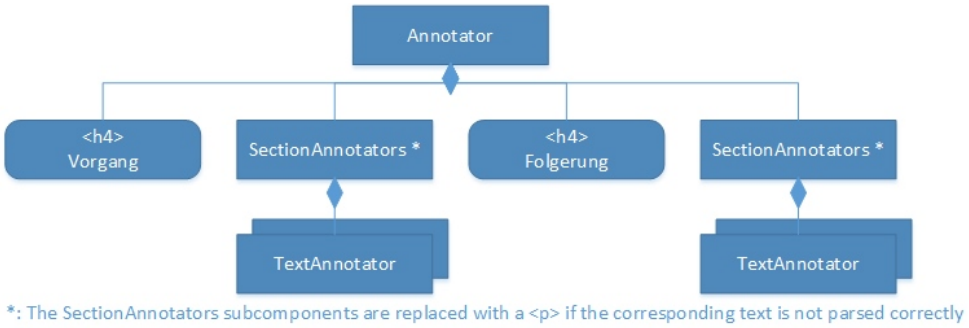

Figure 7.5: Annotator

7.1.1.5 Annotation

This is the Annotator subcomponent.

7.2 Quick API Overview

Endpoint	Description
/getSimilarReportsFromReport	Request a list of similar reports for a given report ID.
/getSimilarReportsFromOrder	Request a list of similar reports for a given order ID.
/semanticFromOrder	Request a list of similar reports for a given order ID using a similarity calculation algorithm based on word embeddings.
/orders/<order_id>	Request information for a given order ID
/reports/<report_id>	Request information for a given report ID or add the report to the knowledge base, associating it with any passed components.
/reports/<report_id>/file	Download the file of the report with the given ID.
/annotations	Request a list of the annotations and categories within the knowledge base or add new annotations (associated with components) to the knowledge base.
/components	Request a list of the components within the knowledge base or add new components (associated with COCs) to the knowledge base.
/query	Request a list of all keywords including their stems, category and component associations from the knowledge base.

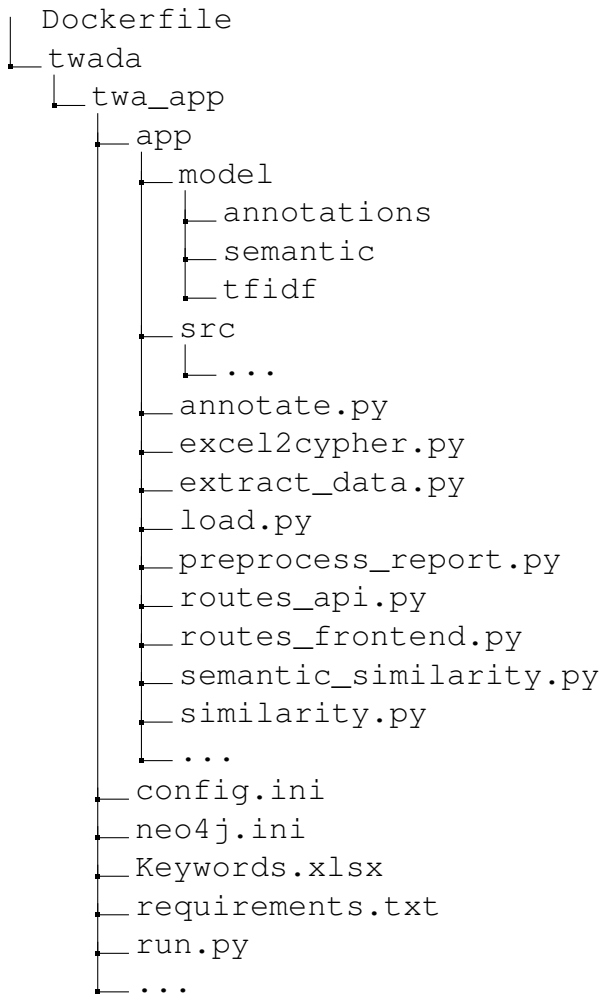

Figure 7.6: Server Directory Tree

7.3 Backend File Structure

Figure 7.6 describes the directory tree of the system. The Dockerfile is in the root folder, as all files must be copied to the Docker image. Inside of ”twa_app”, configuration files (config.ini for the ADM database and neo4j.ini for Neo4j) exist, alongside the Excel file described in Section 4.3.3.1, the requirements.txt file for the installation of all Python packages, and the start script of the server (run.py). The ”app” folder contains the entirety of the backend and frontend code of the server, with all of the frontend code concentrated in the ”src” folder. A model folder contains the generated similarity matrices for each topic (these files had to be redacted due to containing classified TWA report contents). The majority of the Python modules are imported as part of the routes_api.py file.

7.4 Setup of the Development Environment

Requirements:

Python3.7 Docker (for Windows: Docker for Windows) ”Anaconda3-2018.12-Windows-x86_64” ”neo4j-desktop-offline-1.1.17-setup”

Recommendations: Windows 10 IDE, i.e. Visual Studio Code

7.5 Deployment with Docker

Code listing 7.1 explains the steps of the Dockerfile. Note that certain authentication URLs, library sources and filenames containing versions have been redacted for security reasons.

```
1  # Define base image
2  FROM ubuntu:18.04
3  # Declare variables (passed by build script)
4  ARG user
5  ARG password
6  ARG technicaluser_user
7  ARG technicaluser_password
8  ARG ENV
9  # Declare environment variables
10 ENV http_proxy (REDACTED)
11 ENV https_proxy (REDACTED)
12 # Update software repository index
13 RUN apt-get update
14
15 # From http://tlfvincent.github.io/2016/04/30/data-science-with-docker/ ,
16 # install the GNU Fortran Compiler, dependencies, some utilities
17 RUN apt-get install -y gfortran libfontconfig1 --fix-missing
18 RUN apt-get install -y tar git curl wget net-tools build-essential unzip
      libaio-dev
19
20 # From https://gist.github.com/monkut/c4c07059444fd06f3f8661e13ccac619 ,
21 # install Python 3 and utilities
22 RUN apt-get install -y python3.6 python3.6-dev python3-pip python3.6-venv
      swig python-setuptools
23 RUN apt-get install -y git gcc
24
25 RUN python3.6 -m pip install pip --upgrade
26 RUN python3.6 -m pip install wheel
27 RUN python3.6 -m pip install paramiko
28
29
30 # From https://github.com/cmusphinx/pocketsphinx-python/issues/40
31 # and https://github.com/greghesp/assistant-relay/issues/49 ,
32 # install libraries necessary for pocketsphinx, JDK
33 RUN apt-get install -y libpulse-dev libasound2-dev
```

```
34 RUN apt-get install -y default-jdk
35
36 # Install Common Internet File System utilities
37 # See also https://wiki.ubuntuusers.de/Samba_Client_cifs/
38 RUN apt-get install -y cifs-utils nfs-common
39
40 # Create folder for mounting external file system
41 RUN mkdir /mnt/myvolume
42
43 # Install package to mount CIFS directory inside Docker
44 RUN wget (REDACTED)
45 RUN dpkg -i (REDACTED)
46
47 # Install Word extraction tools
48 RUN apt-get install -y antiword
49 RUN git clone (REDACTED) --config http.proxy=
50 RUN python3.6 -m pip install ./textract
51
52 # Copy requirements.txt and change working directory
53 COPY ./twa_app/requirements.txt /code/
54 WORKDIR /code
55
56 # Install the Oracle Instant Client library (multiple subpackages)
57 RUN wget -q (REDACTED)
58 RUN wget -q (REDACTED)
59 RUN wget -q (REDACTED)
60 RUN dpkg -i /code/*.deb
61 RUN rm -rf /code/*.deb
62
63 # Install necessary Python libraries from requirements.txt
64 RUN python3.6 -m pip install -r requirements.txt
65
66 # Define default text encoding to prevent faulty JSON parsing
67 ENV PYTHONIOENCODING=UTF-8
68
69 # Define environment variables for Oracle Instant Client; run linker
70 ENV ORACLE_HOME=/usr/lib/oracle/(REDACTED)/client64
71 ENV LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${ORACLE_HOME}/lib
72 RUN echo "$ORACLE_HOME" > "/etc/ld.so.conf.d/oracle.conf"
73 RUN ldconfig
74
75 # Install packages for diagnosing network problems
76 # also install utlities to fix files directly in the container
77 RUN apt-get install -y iputils-ping manpages vim
78 RUN apt-get install -y smbclient
79 RUN apt-get install -y poppler-utils
80
81
82 # Install NodeJs and the Node Package Manager
83 RUN curl -sL (REDACTED) | bash
84 RUN apt -y install gcc g++ make
```

```
85  RUN apt install nodejs
86  RUN node -v
87  RUN npm -v
88
89  # Backend: Copy generated similarity model files
90  COPY ./twa_app/app/model/ /code/app/model/
91  RUN cd ./app/model/tfidf/; ls
92  # Frontend: Copy static HTML sources
93  COPY ./twa_app/app/static/ /code/app/static/
94  COPY ./twa_app/app/templates/ /code/app/templates/
95  # Backend: Copy configuration files and Excel file with original keywords
96  COPY ./twa_app/*.ini /code/
97  COPY ./twa_app/Keywords.xlsx /code/
98
99  # Frontend: Copy package and package-lock JSON files
100 COPY ./twa_app/app/*.json /code/app/
101 # Frontend: Install node packages
102 RUN cd ./app/; npm i
103
104 # Frontend: Copy source code
105 COPY ./twa_app/app/src/ /code/app/src/
106
107 COPY ./twa_app/app/*.js /code/app/
108 COPY ./twa_app/app/.babelrc /code/app/
109
110 # Frontend: Start build server
111 RUN cd ./app/; npm run build
112
113 # Backend: Copy source code
114 COPY ./twa_app/app/*.py /code/app/
115 COPY ./twa_app/*.py /code/
116 # Disable proxies after installing all packages by overwriting the env vars
117 ENV http_proxy "proxy_overwrite"
118 ENV https_proxy "proxy_overwrite"
119
120 # Now that outside network traffic is disabled, prepare mounting of Windows
      DFS by copying the CIFS mount command to the file system table
121 RUN echo "//(REDACTED).bmw.corp/(REDACTED)$  /mnt/myvolume cifs user=
      $technicaluser_user,username=$technicaluser_user,password=
      $technicaluser_password,domain=(REDACTED),vers=2.1 0 0" >> /etc/fstab
122
123 # Set "ENV" environment variable
124 #  (the value is either "prod" or "dev"; passed by build script)
125 ENV ENV=$ENV
126 # Make run.py file runnable
127 RUN chmod 644 run.py
128 # Mount file systems and start webserver (if successful)
129 CMD ["sh", "-c", "mount -a && python3 run.py"]
```

Listing 7.1: Dockerfile

List of Figures

Bibliography

[1] H. Alani et al. “Automatic ontology-based knowledge extraction from Web documents”. In: *IEEE Intelligent Systems* 18.1 (Jan. 2003), pp. 14–21. ISSN: 1541-1672. DOI: 10.1109/MIS.2003.1179189. URL: http://oro.open.ac.uk/20051/3/Alani-IEEE-IS-2002.pdf.

[2] Giambattista Amati. “Information Retrieval”. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1519–1523. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_915. URL: https://doi.org/10.1007/978-0-387-39940-9_915.

[3] Stephan Augsten. *Was ist Puppet?* June 2018. URL: https://www.dev-insider.de/was-ist-puppet-a-720552/.

[4] Tim Berners-Lee, James Hendler, Ora Lassila, et al. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 28–37. URL: https://kask.eti.pg.gda.pl/redmine/projects/sova/repository/revisions/master/entry/doc/Master%5C%20Thesis%5C%20(In%5C%20Polish)/materials/10.1.1.115.9584.pdf.

[5] Rebecca Bilbro. *Introduction to Document Similarity with Elasticsearch*. Online. June 2018. URL: https://rebeccabilbro.github.io/intro-doc-similarity-with-elasticsearch/.

[6] *Container definition and meaning: Collins English Dictionary*. URL: https://www.collinsdictionary.com/dictionary/english/container.

[7] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Vol. 520. Addison-Wesley Reading, 2010. URL: https://ciir.cs.umass.edu/downloads/SEIRiP.pdf.

[8] Benjy Cui. *rcslider*. Sept. 2019. URL: https://www.npmjs.com/package/rc-slider.

[9] Robin Dunn. *Welcome to wxPython!* Jan. 2019. URL: https://wxpython.org/.

[10] Yoav Goldberg and Omer Levy. “word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method”. In: *arXiv preprint arXiv:1402.3722* (2014). URL: https://arxiv.org/abs/1402.3722.

[11] C. Green et al. “Readings in Artificial Intelligence and Software Engineering”. In: ed. by Charles Rich and Richard C. Waters. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986. Chap. Report on a Knowledge-based Software Assistant, pp. 377–428. ISBN: 0-934613-12-5. URL: http://dl.acm.org/citation.cfm?id=31870.31893.

[12] Kinnary Jangla. “Containers”. In: *Accelerating Development Velocity Using Docker: Docker Across Microservices*. Berkeley, CA: Apress, 2018, pp. 1–8. ISBN: 978-1-4842-3936-0. DOI: 10.1007/978-1-4842-3936-0_1. URL: https://doi.org/10.1007/978-1-4842-3936-0_1.

[13] Anders Janmyr. *A Not Very Short Introduction to Docker*. Sept. 2016. URL: https://blog.jayway.com/2015/03/21/a-not-very-short-introduction-to-docker/.

[14] Wolfgang Kowarschick. *Multimedia-Programmierung*. Hochschule Augsburg, Fakultät für Informatik, gehalten im Wintersemester 2018. Augsburg, 2018. URL: http://mmprog.hs-augsburg.de/.

[15] Lawrence Krubner. *Docker is the dangerous gamble which we will regret*. May 2018. URL: http://www.smashcompany.com/technology/docker-is-a-dangerous-gamble-which-we-will-regret.

[16] Mounia Lalmas and Ricardo Baeza-Yates. “Structured Document Retrieval”. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 2867–2868. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_378. URL: https://doi.org/10.1007/978-0-387-39940-9_378.

[17] Jey Han Lau and Timothy Baldwin. “An empirical evaluation of doc2vec with practical insights into document embedding generation”. In: *arXiv preprint arXiv:1607.05368* (2016).

[18] Mark Leblanc. *Distributed File System - Windows applications*. May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/dfs/distributed-file-system.

[19] Tanner Linsley. *tannerlinsley/react-table*. Sept. 2019. URL: https://github.com/tannerlinsley/react-table.

[20] Garrett Nicolai and Grzegorz Kondrak. “Leveraging Inflection Tables for Stemming and Lemmatization.” In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1138–1147. URL: https://www.aclweb.org/anthology/P16-1108.

[21] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[22] Juan Ramos et al. “Using tf-idf to determine word relevance in document queries”. In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. Piscataway, NJ. 2003, pp. 133–142. URL: https://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/papers/ramos.pdf.

[23] Hassan Saif et al. “On stopwords, filtering and data sparsity for sentiment analysis of Twitter”. In: *LREC 2014, Ninth International Conference on Language Resources and Evaluation. Proceedings.* 2014, pp. 810–817. URL: http://oro.open.ac.uk/40666/.

[24] Robert Sanderson and Paolo Ciccarese. *Web Annotation Data Model*. Online. Dec. 2014. URL: https://www.w3.org/TR/2014/WD-annotation-model-20141211/.

[25] Brandon Satrom. “Choosing the right JavaScript framework for your next web application”. In: *Vitbok RITM0012054. Progress Software Corporation* (2018). URL: https://www.telerik.com/docs/default-source/whitepapers/telerik-com/choose-the-right-javascript-framework-for-your-next-web-application_whitepaper.pdf.

[26] Wikipedia. *Docker (Software) — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 19. August 2019]. 2019. URL: https://de.wikipedia.org/w/index.php?title=Docker_(Software)&oldid=190973702.

[27] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics* 1.1-4 (2010), pp. 43–52. URL: https://link.springer.com/article/10.1007/s13042-010-0001-0.