UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group DICE

Bachelor's Thesis

Submitted to the DICE Research Group
in Partial Fullfilment of the Requirements for the Degree of

Bachelor of Science

A Question Answering (QA) System for the Data Catalog Vocabulary (DCAT)

by
Marten Louis Schmidt

Matriculation Number:
7090644

Thesis Supervisors:
Daniel Vollmers, Adrian Wilke

Thesis Examinors:
Prof. Dr. Axel-Cyrille Ngonga Ngomo,
Jun. Prof. Dr. Henning Wachsmuth

Paderborn, September 13, 2020

Abstract.
More and more government authorities, as well as local authorities, publish data about climate, weather, public transport, etc. as Open Data. The Open Data Portal Germany (OPAL)[a] project aims to aggregate this data to provide a holistic, central access point that can be used by the public as well as by private companies. Currently, the OPAL database contains about 800.000 datasets, which metadata is semantically annotated and described with the Data Catalog Vocabulary (DCAT)[b]. However, in order to be of use, the portal has to be accessible. For people to use it, they first have to know that it exists and what data is available. Question and Answering (QA) systems have proven to be an easy-to-use way to retrieve information from a knowledgebase like the OPAL database. Therefore, to raise more awareness and enable people to learn about the data, I propose a QA system and a social bot that in combination answer natural language questions about the availability and details of the data in the Open Data Portal. A conversational interface like a chatbot has the advantage of being intuitive and engaging at the same time. Additionally, it does not require the user to have any prior knowledge or training, which is fundamental when it comes to reaching as many people as possible.

[a] `https://projekt-opal.de`
[b] `https://www.w3.org/TR/vocab-dcat-1/`

Contents

1 Introduction

The introduction, explains some key concepts that this thesis builds upon and is motivated by. These include the idea of open data, QA systems, the Data Catalog vocabulary and chatbots. After that, an overview of the contributions of this thesis and the structure of the subsequent chapters follows.

1.1 Open Data and OPAL

The Open Knowledge Foundation[Fou] defines openness in the context of data as follows: "Open means anyone can freely access, use, modify, and share for any purpose.". So Open Data is data that is published with the intent that everybody can use it to, among other things, analyze it, gain new insights from it or to build new services with it.

Governments recently have emerged as some of the biggest publishers of open data with initiatives like data.gov[1], data.gov.uk[2] or the EU Open Data Portal[3]. Platforms like these can play a significant role in providing transparency and giving insights into the state of affairs. They can enable the general public to monitor the actions of government bodies and thus encourage more citizen involvement. Jetzek et al. [JAB19] showed in their study how open data from governments could positively influence the generation of sustainable value for the society. Furthermore, Mergel et al. [MKS18] interviewed 15 U.S. city managers who are responsible for open data policy implementations. They concluded that open data platforms not only positively influence internal government processes but also incentive external innovation including novel applications and services.

The Open Data Portal Germany project has the goal to build a platform like this. It strives to centralize data from other existing platforms like mCLOUD[4], European Data Portal[5], and Govdata[6] and seeks to improve on these existing portals by refining the metadata. To demonstrate the usability of the platform and the data, one part of the project is the development of applications including a city app that displays data related to the current location and a chatbot that informs about available data upon questions on social networks like Twitter. This bot and the QA system necessary for its implementation are the main focus of this thesis. In

[1] `https://data.gov`
[2] `https://data.gov.uk`
[3] `https://data.europa.eu/euodp/en/home`
[4] `https://mcloud.de/`
[5] `https://www.europeandataportal.eu/`
[6] `https://www.govdata.de/`

the context of this thesis the term "chatbot" is used to describe the conversational interface for the QA system and not a bot that is designed to simulate a human conversational partner.

1.2 Question and Answering

Question and Answering (QA) means giving short concrete answers to natural language questions. QA systems are programs that are designed to achieve exactly that. There exist two main categories of QA systems. These are text based systems and systems leveraging a knowledge base. This thesis focuses on systems build upon knowledge bases as OPAL encapsulates a knowledge graph which is a type of knowledge base. Most of those QA systems try to find answers by first translating the natural language question to a structured query language like SPARQL and then querying the knowledge base with this translated question. The first step of this process is also called semantic parsing and there exist a variety of approaches on how to do this. One common technique [Ung+12] is to employ templates that have placeholders for predicates and resources. The natural language questions are mapped to a template and the placeholders are replaced with entities and relations from the questions. Other approaches [Hu+18] include training machine learning models to translate questions to queries. However, this requires huge amounts of training data. In Chapter 3 different QA systems are evaluated and considered for an adaptation to the OPAL database and the DCAT vocabulary. The QA system will be the main component of the social bot as it implements the most difficult part, which is answering the users questions.

1.3 Data Catalog Vocabulary

The Data Catalog Vocabulary (DCAT) [Alb+20] is a Resource Description Framework (RDF) vocabulary created to provide better interoperability between data catalogs published on the Web. It helps publishers of open data to annotate their datasets and data services with a standardized metadata vocabulary. This, in turn, can help consumers of the data and allow better aggregation of metadata from multiple catalogs. Furthermore, it can enable searching for datasets across multiple different catalogs with the same query structure. DCAT was standardized in 2014 by the Government Linked Data (GLD) Working Group [Gro14]. The current version is the second iteration of the vocabulary.

DCAT is a fundamental component to this thesis as it simplifies the process of answering questions about the metadata of the OPAL data significantly. Without a consistent vocabulary the QA system would have to incorporate various different formats of metadata annotations. For example, when a user asks a question related to the release date of a dataset, all properties that appear in the database and describe the release date would have to be aggregated, and the data would have to be queried for all of them. If all the data is annotated with DCAT, it only has to recognize the property `dct:issued` and build a query with it.

1.4 Social Bots and Chatbots

Social bots are programs that control a social network account and aid real users on the platform by taking on repetetive task like greeting new users, managing permissions in a chatroom or by providing services like automatically answering frequently asked questions. Bots like the latter are also called chatbots as they provide a chat-based interface for human-computer interactions. In recent years, there has been renewed interest in chatbots in academic and research contexts as well as for commercial use-cases [Dal16]. Another current application case is a coronavirus

symptom checker in form of a chatbot[BG20]. Chatbots in the commercial field often replace human help desks or offer a conversational ordering process. More interestingly, there also have been many successful applications of chatbots concerning open government data [KSV19] as well as open data in general [ANU18] [Jal+20]. Chatbots have many advantages in comparison to conventional user interfaces. Among other things, they are easy to understand and do not require prior knowledge or training. Like Dale said in his article: "Because of its ubiquity, the messaging interface is effectively a frictionless interface[...]" [Dal16]. Furthermore, the capabilities of the bot can be discovered through conversation and do not have to be learned by clicking through complex menus.

1.5 Contribution

The goal of this thesis is to develop a chatbot that answers natural language questions, asked by users in a social network, about metadata in the OPAL database with the aid of a QA system. Therefore, the application will be split into two independent components, the QA module and the bot module. The QA module will implement the QA system that receives questions via an interface and sends back the information it could retrieve from the database as an answer. Therefore, it is independent of a concrete social network or chat tool. The bot module will work as a bridge to one or more popular social networks or messaging platforms such as Twitter[7] or Slack[8]. It will forward questions sent to the account of the bot to the core module and then present the answers to the user.

1.6 Structure of this Thesis

First, an overview of what comparable work has already been published in the field is given. Then the evaluation of different QA systems and if they are suitable for the integration with the chatbot is detailed. Following that, the implementation process of the adaption of the chosen QA system is presented. After that, multiple social networks and how they fit the use case of the chatbot are evaluated. Subsequently, a description of the implementation of the bot module follows. At the end of the thesis, the evaluation of the developed QA system is discussed and a summary of what was done and an outlook of possible future work is given.

[7] `https://twitter.com`
[8] `https://slack.com`

2 Related Work

In recent years, there has been an increasing interest in QA systems, chatbots and in combinations of both.

Chatbots, in general, have many different use-cases and are applied in research as well as commercial contexts. One example is the DBpedia chatbot[ANU18] developed by Athreya, Ngonga Ngomo, and Usbeck which was designed to improve interaction among members of the DBpedia community and also to introduce new users to the DBpedia knowledge graph (KG). It uses the DBpedia mailing list as a source to answer common questions about DBpedia. The data is cleaned, vectorized, and divided into topics which are then used to build conversation rules with RiveScript[1]. Furthermore, the bot is able to answer factual questions by using a combination of WDAqua's QANARY[Bot+16] and WolframAlpha[2]. It distinguishes between three intents of the user: DBpedia Question, Factual Question, and Banter which are identified with RiveScript rules. Each type of question is handled by a separate component. Users can communicate with the bot on multiple different platforms including Slack, Facebook and a custom web interface. Besides trying to answer the users' questions, if the responses contain RDF URIs, the bot enhances them by adding important attributes about the entities to them. The bot also offers follow-up interactions in the form of links to further information and feedback buttons. The QA system and chatbot proposed in this thesis are of similar nature. They connect to several messaging platforms and as adapt an already existing QA solution but focus on the OPAL database instead of DBpedia and its community.

In a follow-up analysis of the DBpedia chatbot[Jal+20], Jalota et al. studied a dataset of the chatbots logs containing 90,800 interactions collected over two years. The average conversation length was around ten messages and from 7561 received feedback messages 3406 were positive and 4155 were negative. The authors divided the data into the categories of request, response, and conversation for an in-depth analysis. They conclude several implications for improvements for future iterations of the bot as well as for knowledge-driven bots in general. Those include among others multilingual support, guiding user input with, for example, auto-correction and guiding the users expectations by highlighting out-of-scope questions. The implemented bot and QA system already feature multilingual support and guide users expectations with a history of answered questions available on Twitter. They could be extended with the other proposed capabilities in the future, cf. Chapter 6.

Lommatzsch and Katins presented a framework for building chatbots tailored to answering questions about public administration services[LK19]. They aimed to improve upon existing

[1] `https://www.rivescript.com`
[2] `https://www.wolframalpha.com/`

approaches by adapting a knowledge base so that it can be used in a natural dialog while avoiding building knowledge bases and training data from scratch. The data the bot queries consists of paragraphs extracted from documents that are organized in a tree structure based on the LeiKa[3] ontology. The query building is supported by multiple annotations of the question including synonyms, geo data, popularity statistics, and translations if they are not German questions. Apart from the main information retrieval components the bot's system also encompasses a dialog state model to handle follow-up questions and understand their context. Furthermore, the bot gathers not only feedback concerning its overall performance but also intermediary feedback to clarify the intent of a question or find out which result fits the best if there are more than one. The authors' evaluation shows a robust performance of the retrieval-based approach and that improvements to the query building could be made based on users' feedback but points out a flawed handling of off-topic questions. The domain of the chatbots featured in this paper are similar to the bot proposed in this thesis as the OPAL database contains data published by various government bodies and administration entities.

The open data chatbot[KSV19] introduced by Keyner, Savenkov, and Vakulenko is a conversational interface to an Open Data database. It utilizes geographic entities to provide users with datasets related to specific locations and topics. Users can engage with the bot in a search and an exploration mode. The first mode is a keyword search which can also include locations. These are answered by the bot with links to datasets related to the keywords. In the second mode the user first chooses a topic from a predefined list and then a geolocation of interest which leads to the bot suggesting datasets from the database related to both. The bot is built with the Rasa framework[4] and its architecture combines different components into one processing pipeline including, among others, an intent classifier, an entity recognizer, and a dialogue state tracker. The intent classification distinguishes between nine intents and is implemented with a support vector machine. For recognizing entities the authors trained a conditional random fields machine learning model. For better geo-entity extraction a look-up table was added. In comparison to the proposed chatbot this bot mostly works with keywords and not full natural language questions. On a different note, many datasets in the OPAL database are also annotated with location metadata which makes geo-entity searches, like presented in the paper, feasible as well.

[3] https://fimportal.de/
[4] https://rasa.com

3 The Question and Answering Module

The QA module is the main component of the chatbot and implements the question answering functionality. It receives natural language questions from the bot module, transforms them into SPARQL queries, sends these to the OPAL database endpoint, and returns the best answers back to the bot module. The module borrows some functionality and ideas from the TeBaQA system which emerged in a following comparison of different QA systems as the best performing one that contains a lot of functions that can be applied to the DCAT vocabulary. The comparison is preceded by an explanation of the scenario and the resulting requirements concerning the QA module. After the comparison comes an in-depth breakdown of the implementation of the module.

3.1 Scenario

This section details the starting point of the QA module development process and the preconditions that influenced the decisions to implement a new system instead of adapting an existing one.

3.1.1 DCAT

The DCAT [Alb+20] itself includes 13 unique classes and 58 unique properties. It furthermore recommends the use of other vocabularies in addition. However, in comparison to the DBpedia ontology with 685 classes and 2795 properties[1] it contains significantly less. Concerning the problem of named entity recognition and determining what metadata properties a question is referring to, the low number of properties could make that process a lot easier. Because there are few classes and properties and their labels and meanings are very distinct, a disambiguation of named entities could proof unnecessary. Moreover, the small number of properties could make it feasible to create specific inference rules. One example would be that the mention of a date would indicate either the property `dct:issued` or `dct:modified`[2], as those are the only ones with date literals as their domain.

To the best of my knowledge there currently exists no QA system specifically for the DCAT vocabulary which makes this the first.

[1] `https://wiki.dbpedia.org/services-resources/ontology`

[2] Table 3.1 lists all namespace prefixes mentioned during the thesis.

Prefix	Namespace
`dcat`	http://www.w3.org/ns/dcat#
`dct`	http://purl.org/dc/terms/

Table 3.1: Overview of mentioned namespace prefixes

3.1.2 Data

As already mentioned in Section 1.1, OPAL accumulates data from multiple different open data platforms like mCLOUD, European Data Portal and Govdata. The OPAL project uses Squirrel[3] as data crawler to gather and analyze the data from the different web sources.

All this data is structured according to the DCAT vocabulary but the vocabularies and entities used for the metadata annotations differ from dataset to dataset. For example, some datasets use the Language Named Authority List[4] from the EU Open Data Portal to declare the language with the `dcat:language` property but others simply use ISO 639-1 strings like "en" and "es" and others again use full language names like "german". Another example are dates which sometimes are formatted as xsdDate[5], sometimes as xsdDateTime[5] and sometimes as string in an arbitrary format. One part of the OPAL project is the removal of inconsistencies and the unification of the vocabularies used for annotations. This simplifies not only searching for specific datasets but more importantly streamlines the named entity recognition part of QA. Moreover, to unify the `dcat:spatial` properties of the datasets the local administrative units[6] published by the European Statistical Office Eurostat are added to the database which makes searching for datasets about specific states or cities easier. It also enables the creation of tools like Show-Geo[7], which displays all available datasets on an interactive map. The data cleaning and metadata refinement is handled by the Catfish component[8] which was developed as part of the OPAL project. More info about what cleaning methods were applied follows in Section 3.4.1.

Figure 3.1 depicts the processing pipeline that transforms the source data to the OPAL graph which is then incorporated as the single source of truth for the QA system. For all the testing and evaluations of the QA system, version 2020-07 of the OPAL graph[9] was used.

3.1.3 Existing Benchmarks and Training Data

Many state-of-the-art QA systems target the DBpedia database and, therefore, can build upon many existing resources like the benchmarks from the Question Answering over Linked Data (QALD) challenges[10] and tools like DBpedia Spotlight[11] for named entity extraction. TeBaQA [Vol+], for example, employs the QALD 8 and 9 datasets, which contain natural language questions and their corresponding optimal SPARQL queries, to extract isomorphic graph patterns from the queries and then generalize those groups of isomorphic queries to classes of templates. Another example is the QA system developed by Lukovnikov et al. which trains a neural network using the Simple Questions dataset[Bor+15] to return the best matching subjects and predicates for a given question.

[3] `https://github.com/dice-group/Squirrel`
[4] `https://data.europa.eu/euodp/en/data/dataset/language`
[5] `https://www.w3.org/TR/xmlschema-2/`
[6] `https://ec.europa.eu/eurostat/de/web/nuts/local-administrative-units`
[7] `https://github.com/projekt-opal/hackathon/tree/show-geo`
[8] `https://github.com/projekt-opal/catfish`
[9] `https://hobbitdata.informatik.uni-leipzig.de/OPAL/OpalGraph/DCAT-QA/`
[10] `http://qald.aksw.org/`
[11] `https://www.dbpedia-spotlight.org/`

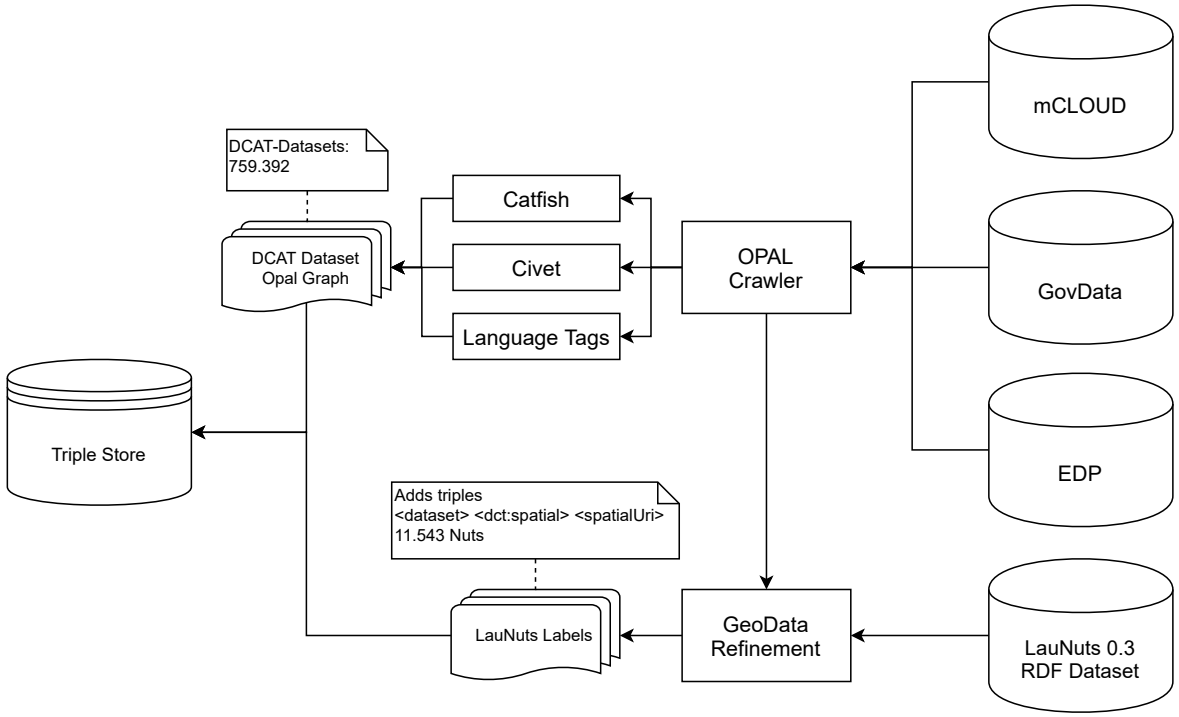

Figure 3.1: Overview of the data processing pipeline.

For DCAT and the OPAL database there are currently no existing question sets or benchmarks like this. This makes a lot of the proposed approaches for QA and the DBpedia database, especially systems that employ machine learning and involve huge amounts of data, infeasible or require a lot of prior work. For example, to apply TeBaQA to the OPAL domain you first would have to manually create a set of optimal question-query pairs that is large enough.

3.2 Requirements

The goal of the QA module, as mentioned before, is to answer natural language questions about the metadata in the OPAL database. More specifically, this results in the requirements which are elucidated in the following section.

3.2.1 Translating Natural Language Questions to a Structured Query Language

To fetch answers the QA module needs to be able to query the data from the OPAL database. The most straightforward way to achieve this is to use a structured query language like SPARQL. To build a SPARQL query the key subjects of the question and their relations have to be extracted from the question. This means in particular:

Recognizing DCAT Properties All properties of the DCAT vocabulary that a question mentions directly and indirectly have to be recognized and considered for inclusion in the final query. For example the question "Which German datasets were issued last?" directly mentions the property `dct:issued`. Also, the word "German" is indirectly referring to the property `dct:language`. Both have to be recognized to correctly answer the question.

Recognizing Entities To answer questions that are more complex than requests for all values of a single property, all mentioned entities and metadata values need to be recognized. To build a complete query for the question "Which German datasets were issued last?" the language entity `http://publications.europa.eu/resource/authority/language/DEU` from the EU language vocabulary[12] has to be recognized.

Recognizing Additional Semantic Properties The QA system should also be capable of answering users asking, for example, for the number of datasets with specific properties or for the latest published datasets. To build queries containing the according aggregation functions the words indicating aggregates like the amount or maximum have to be detected.

Building Query After extracting all relevant properties and entities from a question all these elements have to be combined to build a structured query.

3.2.2 Additional Requirements

In addition to the core requirements for the QA system, there are several functionalities that need to be implemented to integrate the system into the bot application.

Receiving Questions The bot module needs to be able to forward questions from users to the QA system. Hence, it has to offer some form of a communication endpoint.

Fetching Results for SPARQL Query To answer a question with actual results and not just a structured query the QA system has to be able to send a query to the data store, receive the results and forward them back to the bot module.

Answering Questions in Different Languages The QA system should be capable of answering at least German and English questions. It also should be possible to easily extend the system with support for further languages. Multilingual support was one of the key features that Jalota et al. concluded to be beneficial for general knowledge-driven conversational interfaces in their analysis of the DBpedia Chatbot[Jal+20].

3.2.3 Optional Requirements

The following features are not essential to the functionality of the QA system but should nevertheless be considered as they could improve the usability of the bot or benefit future enhancements and extensions of the QA module.

Offering Simple Ways of Extension As mentioned in the section about multilingual support the system should be easily extensible for more languages but also the extension of other components of the QA pipeline should be possible.

Fetching More Results for a Previously Answered Question For a frontend where a user can easily ask for more results to a question, the QA module should offer an endpoint where upon receiving a query it increases the offset in the query and fetches the next x results from the database.

[12] `https://op.europa.eu/en/web/eu-vocabularies/at-dataset/-/resource/dataset/language`

3.3 Comparison of Existing QA Systems

There exist already many QA systems with various strategies. This section presents multiple existing QA systems, details their approaches and determines how they fit the before declared scenario and requirements. At the end, the adaption of one of the presented approaches will be weighed against developing an own solution.

For the comparison these are the main points that are contemplated:

- What components can be ported to the OPAL database?
- Does the approach employ machine learning models that require training data?
- How complex is the approach?
- How is the entity, property, etc. recognition handled?
- Does the approach make use of any external tools like DBpedia's Spotlight?

TeBaQA Vollmers et al. introduced a template-based QA system [Vol+] which uses graph-pattern isomorphisms to learn templates from benchmark datasets. The approach is based on the observation that in a benchmark dataset, like QALD-8[Usb+18], basic graph pattern isomorphisms can be found among the SPARQL queries and that those structurally similar queries represent syntactically similar questions. The questions considered similar based on the isomorphism of their corresponding queries are then grouped together which leads to many classes of similar questions. The queries in a class are generalized to templates. These classes are then used to train a classifier.

The first phase of the proposed QA pipeline is named Question Processing and Information Extraction and is responsible for first identifying all semantically relevant n-grams and then extracting entities, properties and classes from them. In the second phase, Question Classification, the before trained classifier is used to map the question to a class of questions. The template belonging to the class is then filled with the extracted entities, properties, and classes in all possible ways to build queries. These are executed in the next step so that the answers can be ranked in the final phase based on the expected answer type to deliver only the best answer.

As already mentioned in Section 3.1.3, there currently do not exist any benchmark datasets like QALD for the DCAT vocabulary. Therefore, adapting TeBaQA including training a classifier is not feasible. However, other components of the system could easily be adopted. For the recognition of entities, properties and classes, for example, all entities, properties and classes from DBpedia are indexed and all n-grams consisting of up to six neighboring words from the question are searched in the indices. This approach could easily be adapted for recognizing DCAT properties and entities of the OPAL database. Furthermore, if templates are provided in some other way, the injection of entities, etc. could also be inherited, as well as the rating of answers. Functionalities like offering an HTTP endpoint and executing SPARQL queries against an endpoint could also be adopted.

So overall the main idea behind the approach is unfortunately not applicable but nevertheless, because many other components could be reused, the adoption of TeBaQA should still be considered.

WDAqua-core1 In "WDAqua-Core1" Diefenbach, Singh, and Maret [DSM18] presented a new multilingual knowledge base (KB)-agnostic QA system. Their proposed method stems

from the idea that the syntax of a question is less relevant for understanding a question and the semantics alone would suffice to capture the intent of a user.

The conceptual answering process of the system consists of four distinct steps: expansion, query construction, query ranking and answer decision. In the first one, similar to TeBaQA, all n-grams except ones that are stop words are matched against an index of labels of all entities, properties, and classes from the KB. The second step is the most complex one and involves building queries given the found lexicalizations from step one. For this, the system makes use of 4 general query patterns including at most 2 triples. To fill these patterns with valid triples, meaning triples that exist in the KB, the KG is traversed in a breadth-first manner starting from all resources found in step one to a depth of two. In addition, all distances between all visited pairs of nodes are measured and saved. These then serve as input for a recursive algorithm which generates all connected triple patterns with a maximum of K triples so that the query templates can be filled with them. In the next phase, called "Ranking", the constructed queries are ranked according to five features including the number of variables and triples, the relevance of the resources, the edit distances of the labels of the resources and their associated words in the question, and the number of words covered by the query. In the last step, called, "Answer Decision", the system decides if the best answer is returned or not by using a trained model based on logistic regression to predict whether the F-score of the query exceeds a certain threshold. If it does, the answer is returned; if not, no answer is given.

According to the authors, their approach has several advantages over other QA systems including portability to other KBs, portability to new languages and robustness to malformed questions, and keyword questions. The last one results from the disregard of the syntax of the questions whereas the portability properties depend on additional resources, i. e. stop word lists and stemming capabilities for other languages and lexicalizations for other KBs. Moreover, the authors point out that the QA is also capable of querying multiple KGs at the same time.

Their evaluation shows that the proposed system can outperform many other systems especially when it comes to multilingualism and querying multiple KBs.

The way of handling the recognition of entities, properties and classes is very similar to the approach from TeBaQA and could, therefore, be easily adapted to the OPAL KB as well. Additionally, the technique of manually defining templates for queries filling them with triples found in the question would be feasible to adopt as well. However, because the DCAT ontology is not nearly as exhaustive as the DBpedia ontology and others used in the paper and properties, and entities are far less ambiguous, the proposed algorithms for building a query might be inordinate and a simpler solution would suffice. Furthermore, to achieve the performance presented, a model for the ranking of queries needs to be trained. That means that datasets containing questions and corresponding answer sets are required.

QAnswer QAnswer [Rus+15] is a QA system developed and published by Ruseti et al. and focuses on the matching of phrases from the question to entities in the ontology. It employs different approaches for each entity type while using DBpedia and Wikipedia as KB. It was the second best solution of the QALD-5 challenge [Ung+15].

The proposed solution consist of a multi-step pipeline which starts with building a directed graph containing all tokens from the question. These are then annotated with the help of Stanford CoreNLP[13] with lemmas, part-of-speech tags and edges for dependencies.

[13] `https://stanfordnlp.github.io/CoreNLP/`

Additionally, numbers and dates are recognized with the Stanford NER library[14]. The next step is responsible for the matching of DBpedia entities which is done for each entity type individually, namely individuals, types, and properties. The detection of individuals is done by searching all sequences of words from the question in an index containing all DBpedia individuals except ones which where filtered out beforehand because of low relevance measured by the number of ingoing links to the individuals Wikipedia page. Next comes the detection of types. Here the approach is based on the observation that for the majority of individuals the type is mentioned in the first sentence of their Wikipedia article. These mentions are extracted by analyzing the dependency graph of the sentences and are then indexed similar to the individuals. The elements of all graphs resulting from the previous step are matched against the index. The following and most complex step is the property detection. The approach is similar to the type detection as it uses Wikipedia but in this case scours whole articles. More specifically, it extracts sentences containing a label of one of the individuals and a label of one of the properties. Assuming that the paths between subject and object in these sentences is an expression of the property, a search is performed to find out all the words of these expressions from a subgraph of the query graph. Then a score is computed indicating if the types of the entities connected by the expression correspond to types in the database. The highest scoring subgraphs then have to be incorporated into the graphs from the previous step by distributing them to as few graphs as possible while not overlapping. For this, the system calculates all maximum cliques in a graph. During the next step all resulting graphs are scored based on the scores from the previous steps and the highest scoring graph is chosen. In the end a SPARQL query is generated recursively from the graph where the searched variable is determined by analyzing the edge nodes in the dependency graph of the question.

Apart from the detection of individuals, the employed methods can not easily be adapted to the OPAL domain and DCAT vocabulary as the data is not based on Wikipedia and, therefore, scouring Wikipedia articles would probably not be the most efficient way to detect properties and types. Also, the number of properties in the DCAT vocabulary is far less than the number of DBpedia properties so a solution as complex as this is not necessary.

In conclusion, there exist many QA systems with numerous approaches. The ones presented here were selected based on their potential adaptability. Many other solutions rely heavily on machine learning models and data and were, thus, discarded early on. Table 3.2 shows an overview of the components that could be adapted and the ones that could not be adapted. However, none of the presented systems can be fully adopted to the OPAL domain as all of them have components not applicable without benchmark data or other required tools, data or services. This lead to the decision to implement a new QA system while only adopting some functionalities from the existing systems.

3.4 Data Preparation

Before the QA system could be implemented, some preparatory tasks had to be fulfilled. Some were part of the OPAL project and were handled by members of the OPAL team, like the cleaning of the data and the addition of location labels. Other tasks, like the German labeling of DCAT properties and the creation of a benchmark dataset were done by me and are part of this thesis. Nevertheless, all of them will be briefly elucidated in the following.

[14] `https://nlp.stanford.edu/software/CRF-NER.shtml`

QA system	Adoptable Components	Non-Adoptable Components
TeBaQA	• recognition of entities, properties and classes • filling of templates / query building	• training of classifier with benchmark dataset • using said classifier to choose templates for a query
WDAqua-core1	• recognition of entities, properties and classes • manually defining templates	• query building algorithm • training a model to rate constructed queries
QAnswer	• recognition of individuals	• recognition of types • recognition of properties

Table 3.2: Components of QA systems that could and could not be adapted

3.4.1 Data Cleaning

As already mentioned in Section 3.1.2, the data in the OPAL database is not uniformly annotated with the same vocabularies. For example, the `dct:lang` property sometimes points to entities from the European Data Portal vocabulary and sometimes to strings containing the language code. As part of the OPAL project, these annotations were unified. The steps completed include:

- All blank nodes, that are not subject of triples, were removed.
- All non-German or non-English titles were removed.
- All datasets, that do not have a German and an English title were removed.
- All triples with literals that contain no value or are unreadable were removed.
- All `dcat:theme` annotations were replaced with a corresponding theme entity from the European Union (EU) theme vocabulary[15] if they were not already.
- For all `dct:format` annotations new entities with the prefix `<http://projekt-opal.de/format>` were created. For example:
  `<http://publications.europa.eu/resource/authority/file-type/CSV>`
  → `<http://projekt-opal.de/format/csv>`.
  The `dct:format` was chosen to represent the file type because the `dct:mediaType` domains' vocabulary[16] is not expressive enough.
- All dates from the `dct:modified` and `dct:issued` properties were converted to the `xsd:date` format.
- All `dct:title` and `dct:description` annotations were updated with language tags.

[15] `https://op.europa.eu/en/web/eu-vocabularies/at-dataset/-/resource/dataset/data-theme`
[16] `https://www.iana.org/assignments/media-types/media-types.xhtml`

- The `dcat:spatial` annotations were unified by adding local administrative units[17][18] to the database.

All this cleaning tasks are implemented across two different projects. The metadata refinement project[19] is responsible for the adding of language tags and geographic data. The catfish project[20] implements all other mentioned cleaning tasks. Additionally, the quality of all the metadata annotations in the database was measured and the annotations were again annotated with those quality ratings. This is implemented in the civet project[21]. The combined execution of all the mentioned components was handled by the OPAL batch application[22]. The configuration of the batch process that was executed can be found here: `https://hobbitdata.informatik.uni-leipzig.de/OPAL/OpalGraph/latest/2020-07/opal-edp-info.txt`. Figure 3.1 shows what data was processed by what components.

3.4.2 Benchmark Creation

For the evaluation of the QA system a set of 50 questions was created and then annotated with corresponding SPARQL queries and other additional properties using the benchmark curation tool QUANT[Gus+19]. The creation and annotation process will be further explained in Chapter 5.

3.4.3 Labeling of Entities, Properties, Classes

To recognize entities, properties or classes in a natural language question where they are not mentioned directly by their URI, they have to be annotated with labels which can then be searched in the question. As mentioned in Section 3.3, using an automated detection method based on Wikipedia like the one QAnswer uses, does not make sense for the OPAL scenario. The manual annotation and index based detection is far less error prone and more feasible because the number of properties and classes is much smaller in comparison to DBpedia.

The DCAT vocabulary already defines English labels for all properties and classes. In combination with the labels from the DCAT-AP.de vocabulary[23] the QA system should already be capable to match many mentions. To further improve the robustness of the detection, synonyms of the existing labels and expressions with the same meaning were added to the sets of German and English labels. All DCAT properties and classes together with their labels were bundled as one JSON file[24] which makes them easily readable by the QA application and also easily extendable with additional labels for the existing languages and also new languages. The structure of the JSON file can be seen in Listing 3.1.

For most of the entities no manual labelling was required as the vocabularies from the European Data Portal already define German and English labels. So the remaining tasks that had to be completed were parsing the vocabulary rdf files, filtering out all labels in languages that are not required to minimize the file sizes and write them to a new JSON file. This method was applied for the following vocabularies (in parentheses the corresponding property):

- Languages[25] (`dct:language`)

[17] `https://ec.europa.eu/eurostat/de/web/nuts/local-administrative-units`
[18] `https://hobbitdata.informatik.uni-leipzig.de/OPAL/LauNuts/LauNuts-0.3.0/`
[19] `https://github.com/projekt-opal/metadata-refinement`
[20] `https://github.com/projekt-opal/catfish`
[21] `https://github.com/projekt-opal/civet`
[22] `https://github.com/projekt-opal/batch`
[23] `https://www.dcat-ap.de/def/`
[24] `https://github.com/martenls/dcat-qa-system/blob/dev/dcat-qa-system/src/data/dcat+labels.json`
[25] `https://data.europa.eu/euodp/en/data/dataset/language`

- Frequencies[26] (`dct:accrualPeriodicity`)
- Themes[27] (`dcat:theme`)
- Licenses[28] (`dct:license`)
- Launuts[29] (`dct:spatial`)
- File types[30] where the prefix `http://publications.europa.eu/resource/authority/file-type/` of all URIs was replaced with `http://projekt-opal.de/format/` (`dct:format`)

```
{
    "uri": "http://purl.org/dc/terms/modified",
    "labels": {
        "en": [
            "date modified",
            "modified",
            "modify",
            "modification",
            "change",
            "changed",
            "altered"
        ],
        "de": [
            "aktualisierungsdatum",
            "geändert",
            "änderung",
            "modifiziert",
            "modifizierung"
        ]
    }
}
```

Listing 3.1: Labels of the `dct:modified` property as JSON object

3.4.4 Creation of Templates

The QA system builds valid SPARQL queries by filling out templates that have predefined slots for properties, entities and other values. Due to the reason that an automatic extraction of templates from a benchmark question set similar to TeBaQA is not feasible, the templates had to be created manually.

To alleviate this process, it was done after the creation of the benchmark dataset which is described in Chapter 5. The questions annotated with SPARQL queries could then be used as a starting point to get a general idea of what types of templates are needed to answer questions

[26] `https://data.europa.eu/euodp/en/data/dataset/frequency`
[27] `https://data.europa.eu/euodp/en/data/dataset/data-theme`
[28] `https://data.europa.eu/euodp/en/data/dataset/licence`
[29] `https://hobbitdata.informatik.uni-leipzig.de/OPAL/LauNuts/LauNuts-0.3.0/`
[30] `https://data.europa.eu/euodp/en/data/dataset/file-type`

about the DCAT ontology and the OPAL database. For example, the question “Which datasets about Aachen were published in May 2018?” lead to the inclusion of templates containing an interval filter. More specifically, a filter that checks if one variable lies between a specific upper and lower bound. Further template features that were derived from the benchmark questions and their queries include:

- A filter that checks if a variable matches one of the literals from a given array. This was derived from questions like “Which datasets specify "http://www.bista.tg.ch" as the landing page?” were a literal is directly embedded in the question and has to be represented in the query.
- Count queries which count the number of all possible assignments of a variable in a query. This was derived from questions like “How many Datasets are available as PDF?” where the number of distinct datasets which have a distribution available as PDF has to be counted.
- “Order By” modifiers which order the results of a question ascending or descending by the values of one variable. This is required for questions like “What is the largest file in the database?”.

A simple template language was developed to specify the templates. It adheres to the syntax of SPARQL but replaces entities, properties and literals with consecutively numbered slots. For example, the first property in a template is always `<prop0>`. The same holds for entities (`<entity0>`), single literals (`<literal>`), literal arrays (`<literalArray0>`), and interval bounds (`<lbound0>` and `<rbound0>`). All variables are also numbered consecutively as can be seen in Listing 3.2.

The current templates range from containing one property and one entity to four properties and four entities and include variants with all the just mentioned features. All the templates are templates for select-queries. However, during the loading process, templates for ask queries are derived from the loaded templates.

All templates are saved in a simple text file of which an excerpt can be seen in Listing 3.3.

```
SELECT DISTINCT ?var1
WHERE {
    ?var0 <prop0> ?var1.
    ?var1 <prop1> <entity0>.
    ?var1 <prop2> ?var2.
    FILTER ( ?var2 >= <lbound0> && ?var2 <= <rbound0>)
}
```

Listing 3.2: Template with interval bounds

3.5 Implementation

The following section, details the implementation of the QA system which includes the data store for the OPAL data, an index for property and entity recognition, and the QA application. The first part presents an overview of the architecture of the system and how the just mentioned components are connected and the second part goes into detail about the QA process.

```
SELECT ?var0 ?var1
WHERE {
    ?var0 <prop0> ?var1.
    ?var0 <prop1> <entity0>
}
ORDER BY ASC(?var1)
---
SELECT DISTINCT ?var0
WHERE {
    ?var0 <prop0> ?var1.
    ?var1 <prop1> ?var2
    FILTER ( ?var2 IN (<stringArray0>) )
}
---
SELECT DISTINCT ?var1
WHERE {
    ?var0 <prop0> ?var1.
    ?var1 <prop1> ?var2.
    FILTER ( ?var2 >= <lbound0> && ?var2 <= <rbound0> )
}
---
```

Listing 3.3: Excerpt from the templates file

3.5.1 Architecture Overview

The QA module consists of three main components:

- a QA application which is implemented as a Spring Boot application[31] in Java
- an Apache Jena triple store and Fuseki SPARQL endpoint[32]
- an Elasticsearch index[33]

Figure 3.2 shows an overview of the components and how they communicate. As part of the question-answering process, the QA system queries the Elasticsearch index for entities and properties by sending HTTP requests with words from the question to the endpoint the index offers. The Fuseki data store also offers an HTTP endpoint where the QA system sends its, from a question generated, SPARQL queries.

3.5.2 Elasticsearch Index

Elasticsearch is a distributed, open source search and analytics engine and supports, among other types, textual, numerical, geospatial, structured, and unstructured data. It is built on the search library Apache Lucene[34]. The QA system primarily makes use of the indexing of structured documents and the fuzzy search capabilities of Elasticsearch. Furthermore, it profits from the quick response times and the simple integration process into a Java application.

[31] https://spring.io/projects/spring-boot
[32] https://jena.apache.org/index.html
[33] https://www.elastic.co/
[34] https://lucene.apache.org/

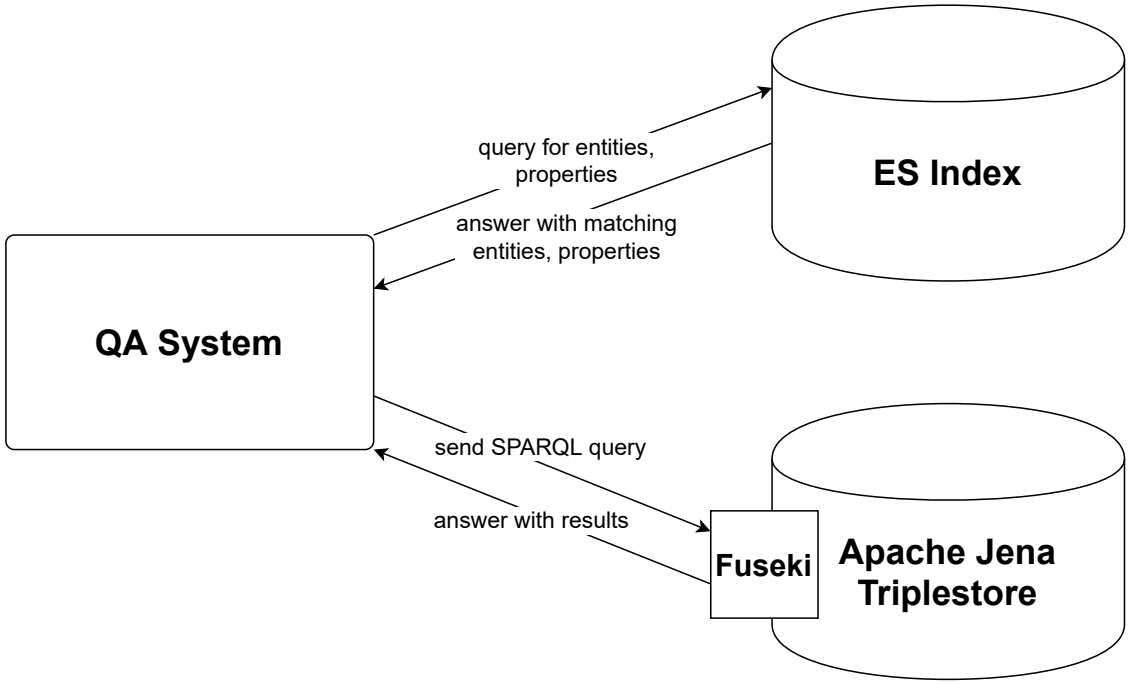

Figure 3.2: Overview of the QA system architecture.

The QA system creates and queries seven indices in total. One for all DCAT properties and the other six for entities, specifically for language, frequency, theme, license, location and file type entities. All of which were annotated beforehand with German as well as English labels as described in Section 3.4.3.

During the startup process of the QA application, an HTTP connection to the Elasticsearch instance is established and the existence of all indices is checked. If it can not be reached, the system retries the connection ten times and shuts down if the last attempt was still unsuccessful. All indices that are not present are then created and filled with documents. In more detail, the JSON files containing the labeled URIs of the respective vocabularies are sent as index requests via HTTP to the Elasticsearch instance. To speed up the process the entities are bundled together in chunks so that one HTTP request carries more than one entity. The whole startup process can be seen in Figure 3.3. During the question answering process the QA system queries the indices in the entity and property detection phase, which are depicted in Figure 3.4. The exact process is further explained in Section 3.5.4.

Since the QA system is already built with the Spring framework, the Elasticsearch library of the Spring Data project[35] is used to implement the just mentioned functionalities. For the deployment of the Elasticsearch instance the official docker image[36] is used.

3.5.3 Apache Jena Triplestore

Apache Jena[37] is an open source Semantic Web and Linked Data framework for Java. It includes, among other components, a persistent high performance triplestore (TDB) and a web server which exposes an HTTP based SPARQL endpoint (Fuseki). The Apache Jena framework is incorporated in the QA system as Docker container based on the `stain/jena-fuseki` image[38]

[35] `https://spring.io/projects/spring-data-elasticsearch`
[36] `https://hub.docker.com/_/elasticsearch`
[37] `https://jena.apache.org/index.html`
[38] `https://hub.docker.com/r/stain/jena-fuseki`

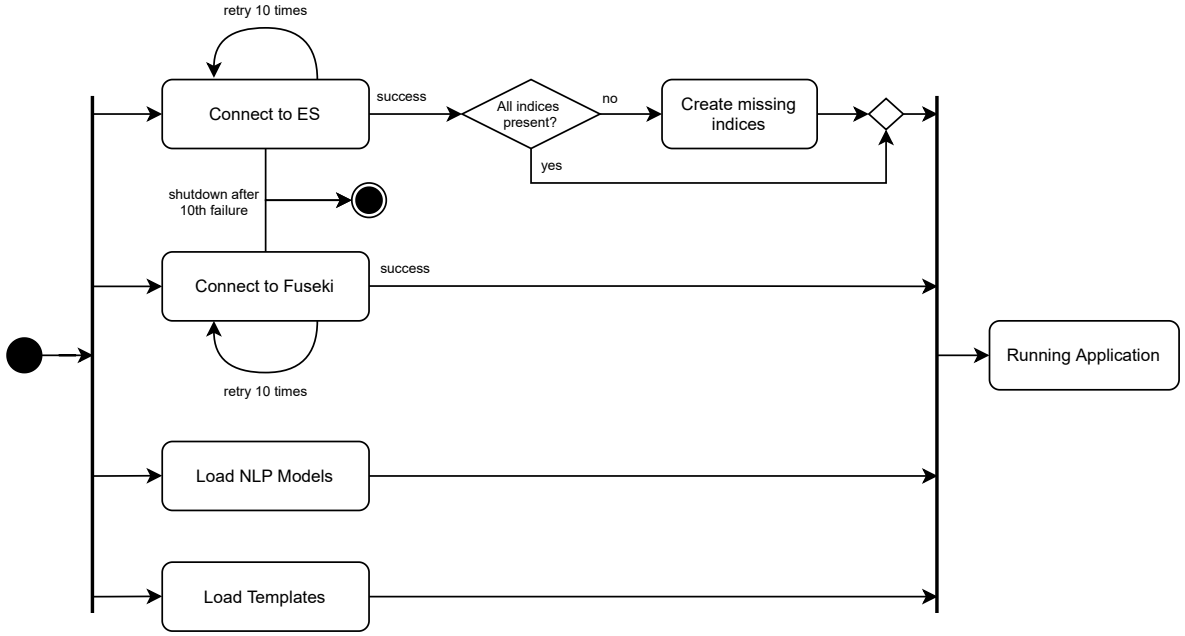

Figure 3.3: Startup process of the QA system application.

which bundles the Fuseki SPARQL server and the TDB triple store. It is responsible for storing all the data discussed in Section 3.1.2 and executing the SPARQL queries generated by the QA application.

The data, which includes the 2020-07 Opal Graph dataset[39] and the geo labels extracted from version 0.3.0 of the LauNuts dataset[40] at the time of the finalization and benchmarking of the QA system, was imported with the `load.sh` script included in the Docker image. More info on how to use this script can be found on the Docker Hub page of the image.

Similar to the Elasticsearch index, during the start of the QA application a connection with the Fuseki instance is established to check if it is available. The same retry mechanism is used as well and displayed in Figure 3.3.

When the connection to the data store is established, the QA application calls the Fuseki HTTP endpoint during the question answering process after the query building phase to execute the built queries. This is implemented with the Apache Jena ARQ library[41].

3.5.4 QA Application

This section will detail the main QA Java application and how it fulfills the core requirements defined in Section 3.2.

The QA application was implemented as a Spring Boot application, which made implementing a lot of the required components a lot easier and more concise in terms of code. It processes questions by annotating a central question object with various different properties, including the detected language, recognized entities and later templates considered fitting for the question. Figure 3.4 gives an overview of the steps involved in the question answering procedure. The process is split into four central phases: Question Processing, Template Selection, Query Building, Result Selection. Those are described in the following.

[39] `https://hobbitdata.informatik.uni-leipzig.de/OPAL/OpalGraph/DCAT-QA/`
[40] `https://hobbitdata.informatik.uni-leipzig.de/OPAL/OpalGraph/DCAT-QA/geo/`
[41] `https://jena.apache.org/documentation/query/index.html`

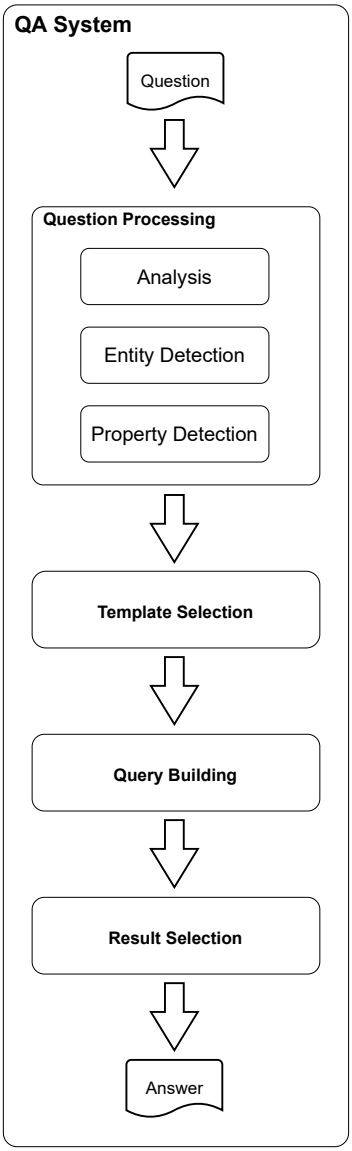

Figure 3.4: Overview of the QA process.

Question Processing

The first step of the question answering pipeline is called "Question Processing" and consists of multiple subphases: analysis and preprocessing, entity detection, and property detection.

Language Detection First and foremost after receiving a question the QA system detects the language of the question sentence. For this the natural language detection library Lingua[42] is employed. It currently supports 74 languages and can accurately detect the language of long sentences as well as short phrases. Furthermore, it does not require complex configuration or the manual provision of trained models like alternatives like Open NLP[43].

Lingua uses a combined approach of a statistical model and a rule-based engine. The rule-based engine first filters out candidates which are considered highly improbable based on

[42] `https://github.com/pemistahl/lingua`
[43] `https://opennlp.apache.org/docs/1.9.2/manual/opennlp.html`

the alphabet of the input text. In the second step a trained probabilistic n-gram model, considering n-grams of sizes one to five, predicts the language of the question. In the current configuration of the QA system Lingua only needs to discern between German and English questions as those are the only languages supported, which improves the accuracy. The language detection is primarily a prerequisite for the following recognition of entities and properties, such that during that process only the labels of the correct language need to be searched.

NLP Annotation The next step of the first phase is the execution of the Stanford Core Natural Language Processing pipeline[44] and, thereby, the annotation of the question with lemmas, part-of-speech (POS) tags, time entities and more. These annotations, as the language detection, are required for subsequent steps which will be explained later.

The Stanford CoreNLP library was chosen to implement this because it supports German and English out of the box for most of the required annotators and can be easily integrated into a Java application. TeBaQA[Vol+] uses CoreNLP as well so most of the implementation details could be adapted from it. Furthermore, CoreNLP needs very little configuration and provides trained models for German and English unlike other libraries like Open NLP.

However, not all annotators support German sentences. For the recognition of German time entities, a new rule set for the SUTime component of CoreNLP had to be created, which will be further explained in the section about entity recognition. Lemmatization, meaning reducing words from their inflected form to their base form, is not supported for German as well and is, therefore, currently only applied for English questions.

Preprocessing The last step of the first phase consists of general preprocessing tasks in preparation for the next steps. Those include splitting the question sentence into single words, filtering out stop words, extracting w-shingles, extracting string literals, and detecting additional properties based on predefined indicators that influence the template selection later in the process.

To split a sentence into single words, first, all punctuation marks and special characters are removed from the question string and then the string is split at all whitespace characters. The resulting list of words is, as all other annotations, added to the question object. For English questions, the CoreNLP library and its tokens and lemmas annotations are used so the list consists only of words in their base form.

Stop words are words filtered out before many common natural language processing tasks and often resemble the most common words in a language. To improve the accuracy of the following entity and property detection these words are filtered out before building n-grams in the next step. To achieve this, all words from the question are matched against a list of stop words. For English questions, a long stopword list from the Ranks NL website[45] is used and for German questions, the list of German stop words from the same website is utilized.

The next preprocessing task is the extraction of neighboring words as n-grams similar to the question processing stage of TeBaQA[Vol+]. N-grams with whole words as items are also called shingles and are defined as sequences of contiguous words from a sample or text. In this case, all n-grams of size one to five are compiled from the question and added to the question object. This again aids the detection of properties and entities. For

[44] `https://stanfordnlp.github.io/CoreNLP/`
[45] `https://www.ranks.nl/stopwords`

example, to answer the question "What datasets exist for Bad Lippspringe?", the entity `http://projekt-opal.de/launuts/lau/DE/05774008` has to be matched. However, if the detection mechanism only considers single words and the index with the location entities is only searched for "bad" and "lippspringe" separately, the entity would not be found. That is why the n-grams have to be included in the search too. The list of n-grams generated by the QA system can be seen in Listing 3.4. The n-gram "bad lippspringe" will then deliver the correct location entity, if searched in the index.

To answer questions that directly mention string literals like the title of a dataset the literals have to be extracted from the question sentence. For this, the literals have to be enclosed in single or double-quotes. The literals are then matched with a regular expression and extracted from the sentence.

In the last step of the preprocessing phase, additional properties of the question are identified. These additional properties signify among other things if the query for the question has to be an ask, a select or a count query, or if the query results have to be ordered and if so descending or ascending. For all of these properties, indicator words and phrases were defined beforehand. For example, the words "is", "are", "exist", "does" and "do" at the beginning of a question sentence indicate a yes or no question and, thus, an ask query.

```
shingles('datasets exist bad lippspringe', 1, 5) =
    [
        'datasets',
        'datasets exist',
        'datasets exist bad',
        'datasets exist bad lippspringe',
        'exist',
        'exist bad',
        'exist bad lippspringe',
        'bad',
        'bad lippspringe',
        'lippspringe'
    ]
```

Listing 3.4: N-grams without stop words for the question "What datasets exist for Bad Lippspringe?"

After the preprocessing, the next phase of the question processing part of the QA pipeline is the entity detection. This phase proceeds differently for two types of entities: the ones represented as URIs with labels that are indexed by the Elasticsearch index and time entities.

Language, Frequency, Theme, License, Location and File Type Entities The process of the labeling and indexing of all these entities was already described in Section 3.4.3 and Section 3.5.2.

The concrete implementation of the entity recognition as part of the QA pipeline comes down to iterating over all n-grams of the question, that were generated in the preprocessing phase, and querying the Elasticsearch index for each of them. The queries are parameterized with the following values: the language of the question, so only labels in the correct language are scoured, the fuzziness, which determines the limit of how far the

edit distance between the n-gram and the label can be to still be considered a match, and the maximum number of results that should be returned by index.

Because the later template selection relies heavily on the number and types of the detected entities and, thus, any superfluous entities that were not exact matches should be dismissed, the fuzziness is never higher than 1. Moreover, the number of maximum results is always set to 1 as there are no mentions which could express two entities at the same time and furthermore no extra disambiguation is applied later so extra results would directly influence the template selection and would be included in the query. Additionally, all duplicates of entities of one type are eliminated, so that for the question "What datasets with the theme education and culture exist?" the theme entity `http://publications.europa.eu/resource/authority/data-theme/EDUC` is only annotated once.

The found entities are all added to the question object separated by their type, to enable the inferring of properties from the types as described in the subsequent section.

Time Entities Time entities are different from location, language, theme, etc. entities because their domain is not defined by a finite vocabulary but rather by the infinite range of a time expression. The approach to label each date and time entity and index them is not feasible, so they have to be detected in another way. The SUTime component, mentioned before in Section 3.5.4, does exactly that. It is a temporal tagger designed to recognize and normalize natural language time expressions and was developed by Chang and Manning[CM12]. The library is built on regular expression patterns and supports four types of temporal objects: time, duration, interval, and set. Because frequency entities are already handled as described in the previous section, the only relevant objects are time and interval objects.

The SUTime library was chosen for this task because the QA application already integrates the Stanford CoreNLP pipeline, which SUTime is part of, and, hence, the implementation effort was much lower than that of the integration of an additional library like HeidelTime[46].

Unfortunately, the library does not support German time expressions out-of-the-box. Thus, a new rule set defining regular expressions adjusted for German time expressions had to be created. However, because English and German time expressions are structure wise pretty similar and recognizing simple expressions is sufficient for QA, the English rule set could be adapted by simply translating all the phrases.

The concrete implementation of the time entity recognition was as simple as adding the SUTime flag to the Core NLP configuration and extracting the entities from the Core NLP annotations generated in the preprocessing phase. To support relative time mentions like "last month", SUTime was configured to resolve relative time expression according to the current date. The extracted time and interval entities are added to the question object to be further processed in the query building phase.

The recognition of time and interval objects is necessary to answer questions which specify values for the `dct:issued` or `dct:modified` properties like the question "What datasets were issued in February 2019?".

The detection of entities is directly followed by the detection of properties.

[46] `https://github.com/HeidelTime/heideltime`

Properties As part of the preparation, the vocabulary of DCAT properties was similarly labeled to the entities as mentioned in Section 3.4.3. Also in conformity with entities, an index for the properties and labels is created during the launch of the QA application cf. Section 3.5.2.

The first step however is not querying the index but inferring properties from the previously found entities. This means that for every found entity of one type the corresponding property is added to the question. For example, for every found location entity the property `dct:spatial` is added and for every found theme entity the property `dcat:theme` is added.

After that, three additional manual inference rules are applied. For the property `dcat:byteSize` special indicators were defined beforehand. This property is barely mentioned directly in questions and is more likely to be asked indirectly with questions like "How big are distributions from datasets about traffic in Berlin?". That is why the property can only be detected by looking for indicators like in this case the beginning of the question with "How big". Furthermore, the approach checks, whether the question indicates asking for the results to be ordered by the `dcat:byteSize` property. For instance, the question "What is the biggest file in the database?" can only be answered by a query that orders the results descending by size. The indicators, in this case, are simply the superlatives "biggest", "largest" and "smallest". The third rule is similar to the second but for the `dct:issued` property. Questions like "What are the latest datasets for Rostock?" require the results to be ordered descending by the publishing date. Therefore, "newest", "most recent" and "latest" are treated as indicators for the `dct:issued` property and a query ordered by the values of that property. The same property is also added at the end of the property detection phase if time or time interval entities are present and neither the property `dct:modified` nor `dct:issued`, which are the only properties with dates as their domain, are already in the list of detected properties. The reasoning for this is that questions for the `dct:modified` property would probably mention it directly and questions mentioning neither of the two but contain time entities probably mean the issuing date like "Are there datasets about the environment from October 2018?".

The last step resembles the process of detecting location, language, etc. entities. The property index is queried for all shingles of the question with a fuzziness of 1 for English questions and a fuzziness of 2 for German questions because the English words are already lemmatized whereas the German words can appear in inflected forms.

Template Selection

The next phase of the QA pipeline is the selection of appropriate templates for the annotated question. This phase builds on all the work done in the previous step by choosing a template based on the found entities, properties and other additional annotations of the question.

The template file created during the preparation phase, cf. Section 3.4.4, is loaded at the start of the QA application and the templates are parsed and their properties, like the number of entity slots, the number of property slots or whether they contain a count operation, are detected. The extraction of properties is done by matching the according keywords with regular expressions. All properties can be seen in Listing 3.5. Furthermore, because the templates are all for select queries, at the same time as those templates are parsed, ask templates are derived from them. This is done by removing the `SELECT` clause completely and replacing the `WHERE` keyword with the `ASK` keyword. Moreover, all `ORDER BY` and `LIMIT` modifiers are removed as well. Listing 3.6 displays one of the `SELECT` templates and the derived `ASK` template side by side.

```
String templateStr;
int propertyCount;
int entityCount;
int literalArrayCount;
int literalCount;
boolean countAggregate;
boolean groupByAggregate;
boolean havingAggregate;
boolean orderDescModifier;
boolean orderAscModifier;
boolean limitModifier;
boolean distinctModifier;
boolean literalArrayFilter;
boolean intervalFilter;
boolean literalFilter;
```

Listing 3.5: All properties that are extracted from a template

```
SELECT DISTINCT ?var1
WHERE {
    ?var0 <prop0> ?var1.
    ?var1 <prop1> <entity0>.
    ?var1 <prop2> ?var2
    FILTER (?var2 IN
    ↪ (<literalArray0>))
}
ORDER BY DESC(?var2)
```

```

ASK {
    ?var0 <prop0> ?var1.
    ?var1 <prop1> <entity0>.
    ?var1 <prop2> ?var2
    FILTER (?var2 IN
    ↪ (<literalArray0>))
}

```

Listing 3.6: A SELECT template and the derived ASK template.

During the selection process of the templates the properties of the templates and all the annotations of the question are combined to form ratings for the individual templates. The rating is an integer value that indicates how well a template fits the question. A higher value means a better fit. The rating is calculated by comparing the template's properties with the question's properties. For every question either all `SELECT` or all `ASK` templates are rated depending on the presence of indicators for an `ASK` query in the question.

The most important factors in the rating process are the number of properties and entities in comparison with the number of slots for properties and entities. Templates that have more slots than that can be filled with the found properties and entities are immediately rejected and given a rating of zero because no valid query could be built with them. If the number of properties matches the number of slots exactly, the rating is increased by 10 points. If the match is not exact and the template does not have enough slots, the rating is increased by 10 minus the difference, i. e. the number of properties that can not be filled in. The same holds for the entities.

The rating is, furthermore, influenced by the additional properties of the question and the templates. The absence of a count modifier in the template combined with the presence of a count indicator in the question leads to a rating of zero. The same holds for a template that

has a modifier paired with a question that has no indicator.

The influence of filters in the template and the entities they have to be filled with is not as strict. If a template has a filter slot but the question no values to fill it with, the template is rated with zero. If the question has values for a filter slot but the template does not have the according filter, the rating is decreased by 10 points. If both filter and entities are present, the rating is increased by 10. In the case of `ORDER BY` modifiers the rating is increased by 10 points if the presence of the modifier and the indicator match and decreased if they do not.

After all templates have been rated, the highest rating is determined and then all templates with this rating are passed on to the query builder. If there are more than five templates that share the highest rating, only the first five templates are passed on.

Query Building

In the query building phase the prior selected templates are filled with the entities, properties and values the question is annotated with. The technique applied for this is first filling the templates in every possible way and then pruning queries that are not valid or do not make sense. In detail the query builder executes the following steps for each template:

1. Fill the property slots with all possible combinations and permutations of properties.
2. Fill the entity slots that are not part of filters with all possible combinations and permutations of entities and immediately discard the partial queries were an entity is not in the domain of its corresponding property.
3. Fill the literal array slots with all possible combinations and permutations of literals.
4. Fill in the left and right bound slots of filters with interval entities while converting the dates to the `xsdDate` format[47].
5. Remove queries that order by the wrong property. This decision is made based on the additional question properties `ORDER_BY_BYTESIZE` and `ORDER_BY_ISSUED` that are detected like the other additional properties during the preprocessing phase, cf. Section 3.5.4.
6. Add a limit modifier to limit the results of the query to one if the query has an `ORDER BY` modifier and the question contains the singular verb “is” or “was”. This is required for questions asking for singular superlatives like “What is the biggest file?”.
7. Discard all queries that still contain unfilled slots and are, thereby, not valid.
8. Add all remaining queries to the question object.

After this process is completed for all the prior chosen templates, the question is annotated with multiple valid queries that all could potentially lead to the correct answer.

Result Selection

The final phase is responsible for selecting a single final query and its results. However, to select a result, first, all query candidates from the previous step have to be executed. The implementation of the query execution was already briefly explained in Section 3.5.3.

After all query candidates are annotated with the results from the Fuseki endpoint, a final query and its results are chosen by comparing the type of the answer with the question and

[47] `https://www.w3.org/TR/xmlschema-2/`

prioritizing positive results. If the question indicates an `ASK` query all query candidates without a boolean value in the answer are removed and the first positive result and query pair is picked if there is one. If not, the first pair in the list is simply chosen. In the case of a question indicating a `SELECT` query all candidates without a result in the form of variables and mappings are discarded. Then, it is distinguished if the question demands a count query or not. If it does, the integers from the results are extracted and the pair with the highest count result is returned as answer. For standard `SELECT` query questions the candidates with no results are discarded and then the first one in the list with results is picked. If none of the candidates have results, no answer is returned.

The described question and answering procedure fulfills all the requirements stated in Section 3.2. This can be seen again in Figure 3.5 which depicts the QA process by example. It implements the detection of properties, entities as well as additional semantic properties and combines all the information found to build a structured query. Furthermore, the additional requirements are satisfied as well. The execution of SPARQL queries was covered in Section 3.5.3 and how the system manages the answering of German as well as English questions was explained throughout the account of the QA section. The third additional requirement is the communication between the QA system and the bot module. For this a simple HTTP endpoint was implemented. It expects the question as path parameter of an HTTP GET request and answers with a JSON object containing the SPARQL query built by the QA system and its results formatted as SPARQL JSON Results[48]. The API specification can be found in Listing 3.3.

Section 3.2 specified in addition to the requirements, that are essential for the bot's functionality, two optional requirements. If and how these were realized, is explained in the following.

The first optional requirement concerns the extensibility of the QA pipeline in terms of additional components aiding the QA process as well as in terms of multilingual support. Extending the QA pipeline can be easily achieved by adding more annotators to the project that annotate the question object with additional properties. Those can be facilitated during later steps of the pipeline like the template selection or the query building phase. If the annotator depends on other annotations, the order in which the annotators are called has to be kept in mind. Regarding multilingual support, the QA system is designed to be easily extensible with more languages, but because almost all the components require language-specific configuration or data the process is still quite arduous. Things that have to be added to support an additional language are:

- labels for all DCAT properties in the new language
- labels for all entities including: languages, frequencies, themes, licenses, launuts, file types
- a list with stop words
- indicators for the additional properties
- a rule set for SUTime or another library that support the detection of time entities in the language
- trained models for the Core NLP pipeline for lemmatization (not essential) or a different NLP pipeline with support for the language

A more detailed guide on how and what components have to be adjusted is included in the documentation of the QA application.

[48] `https://www.w3.org/TR/rdf-sparql-json-res/`

<table>
<tr><td>Path</td><td colspan="2">GET <code>/qa?question={question}</code></td></tr>
<tr><td>Parameters</td><td colspan="2">question (string): the natural language question that should be answered</td></tr>
<tr><td rowspan="2">Response</td><td>200 - OK</td><td>if question could be answered
Body structure:
<pre>{
  "query" : "SELECT ?var1
  ↪ ...",
  "answer" : {
    "head" : {
      "vars" : ["var1"]
    },
    "results" : {
      "bindings" : [{
        "var1" : {
          "type" : "uri",
          "value" : "..."
        }, ...
      }]
    }
  }
}</pre></td></tr>
<tr><td>500 - Internal Server Error</td><td>if question could not be answered</td></tr>
</table>

Table 3.3: Specification of the QA API endpoint

The second additional requirement was an endpoint where more results for a specified query could be fetched. This is required for the functionality of the bot module, that enables the user to ask for more results. It was implemented as HTTP endpoint that expects the SPARQL query as parameter and then increases the offset in the query, executes the query against the Fuseki endpoint, and sends the results back as response.

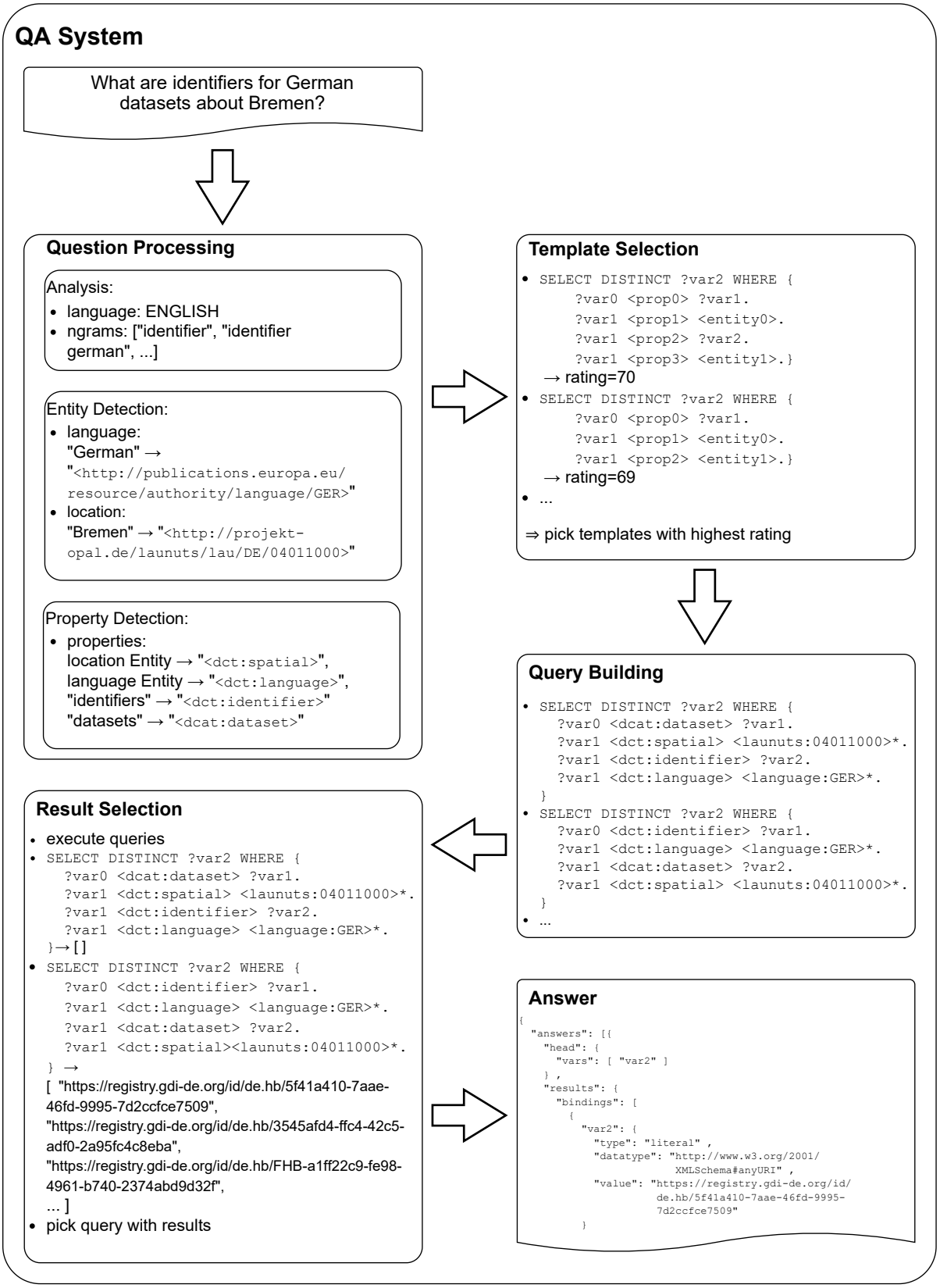

Figure 3.5: Example of the QA process.

4 The Bot Module

The Bot Module implements a Social Media accessor component that builds a bridge between the QA system and one or more Social Network (SN)s or messaging platforms. This connection allows users of the respective platform to send questions to the QA system through a common chat interface. On the side of the SN or messaging platform this interface appears as a "robot" or "bot" user with which can be interacted with like a regular user. The workings and appeal of a so-called Social Bot were already explained in detail in Section 1.4.

In the following, first the reasoning behind the selection of a SN platform for the bot is elucidated and then an overview of how this module is implemented and the design decisions made during that process is given.

4.1 Comparison and Evaluation of Messaging Platforms

There exist many SNs and messaging platforms that provide the Application Programming Interface (API)s that are necessary for development of a bot and could, therefore, be employed as User Interface (UI) component of the bot module. However, the platforms and their specific structure and interpretation of Social Bots differ in terms of how well they are suited for the intended use cases of this bot module. For example, the social network Twitter[1] may offer better bot visibility and discovery options than the collaborative instant messaging platform Slack[2].

The considered platforms include Twitter[3], Telegram[4], Slack[5], and Facebook Messenger[6]. Additionally, the option of building a simple website that allows chatting with the bot is evaluated.

The main metrics that are contemplated are:

- What tools and libraries are offered by the platform which help to integrate a bot?
- What is the estimated complexity of the implementation?
- How big is the user base of the platform?

[1] `https://twitter.com`
[2] `https://slack.com`
[3] `https://twitter.com`
[4] `https://telegram.org/`
[5] `https://slack.com`
[6] `https://www.messenger.com`

- What tools are offered by the platform that help boost the discoverability of the bot?
- How good is the usability of the platform with a focus on communication with bots?

Twitter Twitter[7] is a microblogging and social networking service. Users can post, like, reply to, and repost messages ("tweets"). Twitter also offers a private one-to-one messaging service called Direct Messages. The Twitter API[8] offers endpoints which enable the development of bots that not only communicate through Direct Messages but also tweets.

Enabling users to communicate with the bot over public tweets has the benefit of better discoverability as well as better usability. After some users have asked questions they and their answers from the bot appear publicly on the site of the bot. This has the effect that new users can more quickly grasp the purpose of the bot and what type of questions it can answer. Furthermore, the main goal of the bot, which is to raise awareness and inform about the available data in the portal, can be facilitated by the public availability of the asked questions and answers. So, the more people discover and ask questions, the more attention the bot will draw.

This approach allows to be complemented by the option to also chat with the bot through direct messages which would help users that do not want to publish their questions publicly on their profiles.

One apparent downside of Twitter and all other SN platforms is that an account is required to interact with the bot.

Regarding the complexity of a bot implementation interfacing with Twitter, there exist no mature libraries that use the webhook endpoints from the account activity API. That means that extra effort is required to develop a bot that receives all events in realtime and does not rely on polling.

Telegram Telegram[9] is an instant messaging and voice over IP service. Users can communicate in private one-on-one chats as well as group chats. Telegram offers an extensive bot API[10] that enables developers to create bots that can be messaged directly or invited into group chats.

There exist multiple Telegram API libraries for different programming languages which simplify the creation of a Telegram bot. Examples are TelegramBots[11], python-telegram-bot[12] and node-telegram-bot-api[13]. So in comparison to Twitter, the implementation of a Telegram bot could prove less demanding.

However, Telegram lacks options of discoverability as the bot accounts can not be found as easily as on Twitter and have to be manually invited into group chats. Furthermore, the discoverability is also limited by the total user count of the platform especially when compared to the Facebook Messenger which has more than six times more monthly active users as of October 2019 [Sta].

All in all, Telegram meets all technical requirements to build a bot on its platform but has some disadvantages in comparison to Twitter and Facebook Messenger.

[7] `https://twitter.com/`
[8] `https://developer.twitter.com/`
[9] `https://telegram.org/`
[10] `https://core.telegram.org/bots/api`
[11] `https://github.com/rubenlagus/TelegramBots`
[12] `https://github.com/python-telegram-bot/python-telegram-bot`
[13] `https://github.com/yagop/node-telegram-bot-api`

Slack Slack[14] is an instant messaging and team collaboration platform. It offers so-called "Workspaces" where members can join and communicate with each other via public channels, private channels or direct messages. Public channels are open to everyone in the Workspace whereas private channels can only be accessed by certain members. Direct messages can be sent to all members of the same workspace and also specifically to bot accounts. Slack offers extensive support for integration with third-party services including support for bots[15].

For Slack there exist many community-created libraries[16] written in various programming languages that make developing bots easier.

Users that want to communicate with the bot first have to find and join a workspace where the bot is registered. As nobody knows about the bot in the beginning, it would only be available in a specifically created OPAL workspace. This is a disadvantage in terms of discoverability and also usability. However, integration with Slack may prove useful in the future if communities that are interested in OPAL are communicating over Slack and would like to take advantage of the capabilities of the bot.

Facebook Messenger Facebook Messenger[17] is a messaging app developed by Facebook, Inc and is part of the Facebook SN but is also available as standalone app and separate website. Users can send messages, photos, videos, etc. in private as well as group conversations.

Facebook provides comprehensive resources for bot-development and the integration of the Facebook API[18]. There are official as well as third-party libraries and Software Development Kit (SDK)s to aid developers[19].

Facebook makes multiple ways available to present your bot and let it engage with users[20]. For example, your bot can be linked to a Facebook page you own so that every time a user visits that page they are greeted by the bot and can communicate with it. Other ways include a chat plugin for websites, ads on Facebook that link directly to the messenger or the messenger search where users can discover bot users.

In conclusion, Facebook Messenger is the most used platform among the here presented services and also offers many bot discovery options but lacks the unique features Twitter offers.

Web App The fifth and last option in consideration to enable users to chat with the bot is to develop a simple chat app. This would have the advantage that no account or sign-up is necessary and users can simply visit a website to instantly ask the bot questions.

The ways of discovery would be limited to other websites mentioning the chat app, for example, the official OPAL website. However, this service would be easier to use than all other services and would be suited best for demonstration purposes.

The engineering of a web app like this could prove quite arduous but, as with every before mentioned platform, there exist already many resources. For example, the bot framework Botkit[21] has a simple chat app already included which could be adopted very easily when using this framework.

[14] `https://slack.com/`
[15] `https://api.slack.com`
[16] `https://slack.com/community`
[17] `https://www.messenger.com/`
[18] `https://developers.facebook.com/docs/messenger-platform/`
[19] `https://developers.facebook.com/docs/apis-and-sdks/`
[20] `https://developers.facebook.com/docs/messenger-platform/discovery/`
[21] `https://botkit.ai/`

So all in all, a simple web chat app could be very useful as an addition to the integration with one of the aforementioned services.

As Table 4.1 shows, Twitter has the most advantages over all other platforms and only one disadvantage that it does not share with other messaging platforms (excluding the web app). Furthermore, this disadvantage only influences the complexity of the development and does not have a direct effect on the end product i. e. the user experience. That is why Twitter was chosen as the primary candidate for the integration with the bot. As Facebook ranks second in the comparison it was chosen as an optional platform to integrate if the time allows it. Additionally, due to the trivial implementation (see Section 4.3) and the mentioned upsides in comparison to all SNs, the web app will be implemented as well.

Service	Pro	Contra
Twitter	• users can see question history • public as well as private chat • discoverability through a public account • usability advantage through question history	• account required • no libraries for the account activity API
Telegram	• many API libraries	• account required • only private and group chat • no discoverability • not as popular as FB Messenger
Slack	• many API libraries	• account required • only private and group chat • no discoverability • huge setup effort required
FB Messenger	• many API libraries • very popular • many discovery options	• account required • only private and group chat
Webapp	• many libraries • no account required • good for demonstration	• no discoverability

Table 4.1: Pro and Contra Comparison of Messaging Platforms.

4.2 Goals and Requirements of the Bot Module

In this section, the goals that the Social Media accessor component should achieve and the resulting requirements concerning the implementation of it are covered.

4.2.1 Main Goals

The primary goal of the bot module is to allow users to communicate with the QA system through a simple and easy-to-use chat interface. Particularly, the user should be able to ask a natural language question by sending it as a text message to a bot account which in turn should pass the question to the QA system and present a part of the results in a readable manner back to the user. Additionally, it should offer the user options to ask for more results. If the QA system can not answer the question, the bot should tell the user that it was not able to find an answer and that the user may reformulate the question or ask a different one. Furthermore, when greeted the bot should respond with a greeting and a simple explanation of its capabilities. Concerning the implementation the goal is to provide an application that is easy to deploy, easy to maintain, and which functionalities can be easily extended.

4.2.2 Requirements

The previous stated goals of the bot module result in specific requirements concerning its implementation.

Receiving Messages from Users To answer the questions of the users, the bot module needs a way to access them. In the case of Twitter this means that the tweets as well as the direct messages need to be forwarded from the platform to the application. This also holds for the web app where the messages that a user enters on the website need to be sent to the bot application.

Fetching Answers to Questions In order to send questions to and fetch answers as well as more results from the QA module, the bot module needs to implement functions that consume the API offered by the QA module. This should be implemented independently of the messaging platform so it can be reused for others platforms.

Forwarding Answers to Users To present the answers from the QA system to the users, the bot module application needs a way to send messages back to Twitter and the web app respectively. Twitters API offers endpoints to send tweets as well as direct messages which need to be consumed by the application. The web app has to offer functions that enable sending messages to the user as well.

Managing Conversations The bot module should also be capable to distinguish whether a user is saying hello or if they are asking a question. Furthermore, it should be able to introduce itself, provide information about the OPAL project, and open data in general and the themes of the available data. It should also be able to give example questions. If the QA system is unavailable or can not answer the question it should react accordingly by informing the user.

Figure 4.1 shows a comprehensive overview of the required components and their interactions ordered by time.

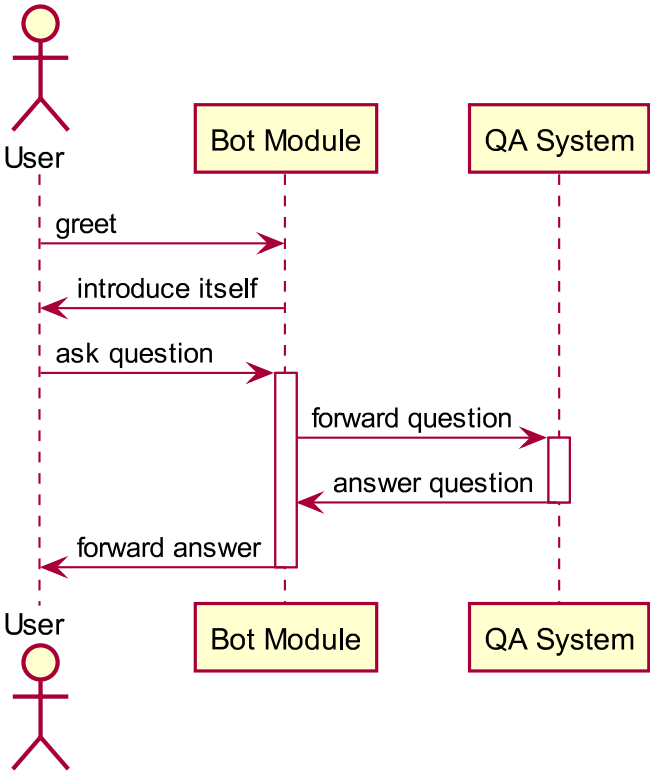

Figure 4.1: Bot module sequence of Events.

4.3 Implementation

This section describes the implementation of the bot module in detail, including which components are responsible for fulfilling each of the before-mentioned requirements. Figure 4.2 gives an overview of all the implemented components of the bot module and how these components communicate with each other.

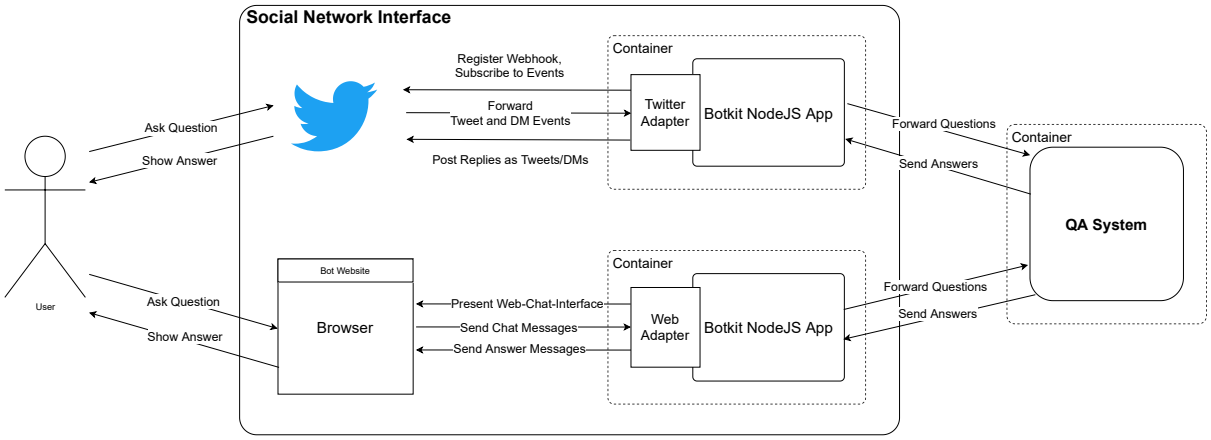

Figure 4.2: Overview of the components of the bot application.

4.3.1 Bot Frameworks

Ideally, a bot framework should offer tools that simplify parts of the bot development process like defining conversations and consuming the APIs of different messaging platforms. Botkit[22] is such a framework. However, Botkit is not the only bot framework. For example, Google's Dialogflow[23] is an extensive framework for building voice and text-based conversational interfaces, but is more complex than Botkit and tied to the Google Cloud Platform making it not the ideal candidate. The wit.ai framework[24] is another example that helps create text or voice-based bots but focuses on integrating with home automation hardware and wearable devices. Microsoft offers the Microsoft Bot Framework[25] which is intended to help developers build, connect, publish and manage chatbots. This framework is build on top of Botkit and adds various tools to it as well as connects it with different Microsoft Azure services ranging from natural language processing pipelines to knowledge stores. As these features are not necessary for implementing the bot module, using this framework would only add superfluous bloat and complexity to the application.

Coming back to Botkit, Listing 4.1 shows how a simple dialog behavior can be defined with it with 10 lines of code. The abstraction of the communication with the APIs of different platforms does not only lead to the code being more concise and easier to understand but also makes it easier to connect more platforms to the bot. Botkit already offers integrations with many messaging platforms like Facebook Messenger, Google Hangouts, and Microsoft Teams.

Furthermore, Botkit also supports the integration of plugins or middlewares that can intercept the processing of messages to, for example, annotate them with further information or to simply log them. There exist many plugins that extend the functionality of the core library with features like natural language processing of messages, storage of conversations, and extraction of statistics from logs.

Because Botkit is a Node.js library, the bot applications using it are preferably built as Node.js applications. Those can be effortlessly packaged and deployed as a container for example as Docker container[26].

All in all, Botkit facilitates many components of bot development and is a well tested, extensible framework that can be easily deployed. These properties meet almost all prior defined requirements. However, there is no existing integration with Twitter so that, in order to integrate the bot with the in Section 4.1 chosen platform, a Twitter adapter for Botkit needs to be implemented. On the other side Botkit already comes with a simple chat web app that can be connected to the bot application which again means less development effort.

In conclusion, Botkit fits the overall scenario best and that is why it was chosen as basis for the development of the bot module.

[22] `https://botkit.ai/`
[23] `https://dialogflow.com/`
[24] `https://wit.ai/`
[25] `https://dev.botframework.com/`
[26] `https://nodejs.org/en/docs/guides/nodejs-docker-webapp/`

```
const qa = new BotkitConversation('qa', controller);
qa.say('Hello!');
qa.ask('What questions do you have?', async(question, convo, bot) => {
    const answer = await fetchAnswer(question);
    convo.setVar('answer', answer);
}, 'question');
qa.say('{{vars.answer}}');
controller.addDialog(qa);

controller.hears('Hi','message', async (bot, message) => {
    await bot.beginDialog('qa');
});
```

Listing 4.1: Defining a simple QA dialog in Botkit.

4.3.2 Botkit Twitter Adapter

This section will detail the implementation of the Twitter adapter for Botkit.

Webhook Registration and Activity Subscription

As mentioned before, to receive messages and tweets from Twitter the bot module application needs to consume the account activity API from Twitter. This API is webhook-based which means that the bot application needs to offer an Hypertext Transfer Protocol (HTTP) endpoint where, after a successful registration, Twitter will send all events tied to a specified account right when they happen. This enables the bot to answer almost instantaneously and eliminates the need for expensive polling mechanisms.

Figure 4.4 shows how the registration of the webhook and the following posting of events takes place. First, the adapter sets up a web server which is publicly available on the internet and listens for incoming Twitter webhook events at the specified Uniform Resource Locator (URL), for example "https://mydomain.com/webhook/twitter". Now the adapter needs to proof that it is both the owner of the app, that is the app created in the Twitter Developer Portal [27], and the owner of the specified webhook URL. To do this the adapter solves a Challenge Response Check (CRC) issued by Twitter upon registering the webhook. This involves Twitter making a GET request of the web app with a unique `crc_token` as parameter and the web app responding with an encrypted `response_token` based on the `crc_token` and the app's consumer secret. If the `response_token` is correct Twitter will register the webhook and respond with the id of the webhook. After that, the adapter subscribes to all account activity by issuing a HTTP POST request to the subscriptions endpoint of the Twitter API. This leads to Twitter sending all events related to the bot account to the webhook endpoint including all tweets from accounts that are followed by the bot account, all tweets mentioning the bot account, and all direct messages sent to the bot account.

Event Processing and Forwarding

When the adapter receives the events from Twitter, it first filters out all irrelevant events like follow and retweet events. Currently, only tweet and direct message events are being processed as those are the only ones necessary to provide the basic functionality defined in Section 4.2.

[27] `https://developer.twitter.com/`

Before the events are passed to the main Botkit logic, they are transformed into `Activity` objects which are more general representations and the basic communication type of the Botkit framework. Listing 4.2 shows a reference `message_create` event object and Listing 4.3 shows how an object like that is translated to an Activity object. The forwarding to the main Botkit logic is done by creating a new `TurnContext` object and calling the `runMiddleware()` method inherited from the `BotAdapter` class with the `TurnContext` as parameter as can be seen on line 27 and 28 of Listing 4.3.

```
1  {
2      "type": "message_create",
3      "id": "954491830116155396",
4      "created_timestamp": "1516403560557",
5      "message_create": {
6          "target": {
7              "recipient_id": "4337869213"
8          },
9          "sender_id": "3001969357",
10         "source_app_id": "13090192",
11         "message_data": {
12             "text": "Hello World!",
13             "entities": {
14                 "hashtags": [],
15                 "symbols": [],
16                 "user_mentions": [],
17                 "urls": []
18             }
19         }
20     }
21 }
```

Listing 4.2: Reference `message_create` event as JSON object.

API Methods and Security

The implemented `TwitterAPI` class of the adapter exposes preconfigured methods to send HTTP GET, PUT, POST and DELETE requests to the twitter API endpoints. Preconfigured means that only the path, the required authentication type, the payload type and the payload needs to be specified and the according headers and authentication tokens will be filled in automatically. This makes sending a direct message as simple as seen in Listing 4.4. Regarding authentication, upon instantiation of the `TwitterAPI` class full Twitter credentials have to be provided, meaning `access_token`, `access_token_secret`, `consumer_key` and `consumer_secret`. Only then all Twitter API endpoints can be accessed. The `bearer_token` required for some endpoints is automatically fetched when needed.

4.3.3 Botkit Web Adapter

Botkit provides an adapter including a simple chat interface implemented as web app out of the box, which can be easily extended and modified. Because, for the purposes of this bot, the

```
private async processSingleDM(message: any, logic: any): Promise<void> {
    // filter out messages sent by the bot
    if (message.message_create.sender_id != this.user.id_str) {
        const activity: Activity = {
            channelId: 'twitter',
            timestamp: new Date(),
            // @ts-ignore ignore missing optional fields
            conversation: {
                id: message.message_create.sender_id
            },
            from: {
                id: message.message_create.sender_id,
                name: message.message_create.sender_id
            },
            recipient: {
                id: message.message_create.target.recipient_id,
                name: message.message_create.target.recipient_id
            },
            channelData: message,
            type: ActivityTypes.Message,
            text: message.message_create.message_data.text
        };
        for (const key in message.message_create.message_data.entities) {
            activity.channelData[key] =
            ↪ message.message_create.message_data.entities[key];
        }

        const context = new TurnContext(this, activity as Activity);
        await this.runMiddleware(context, logic);
    }
}
```

Listing 4.3: Transformation of direct message event into activity object.

existing functionalities suffice, it will only be explained briefly how this adapter and the web app work.

The web app communicates with the web adapter via a websocket and simply forwards all messages the user enters. The web adapter then processes these messages similar to how the Twitter adapter does. They are converted to `Activity` objects and passed to the main Botkit logic. Answers from the bot are then sent over the same websocket connection to the web app, which displays them in the chat interface. The web app also has support for quick replies. An example can be seen in Figure 4.7

Figure 4.5 shows the chat interface of the web app including an example introduction conversation flow were the user first greets the bot, then asks for the explanations of bot OPAL and open data, then inquires the list of themes and at the end wants to see an example question.

```
const res = await this.api.post('/direct_messages/events/new.json',
↪  AuthType.USER_CONTEXT, message);
```

Listing 4.4: Sending direct message via HTTP POST request.

4.3.4 Connection to the QA System

As stated in the requirements in Section 4.2.2 and already shown in Figure 4.1 and 4.2, the bot module needs to communicate with the QA system. The necessary communication can be achieved with one simple HTTP get request. Using the axios library[28] for this leads to a concise implementation as can be seen in Listing 4.5. The answers of the QA systems are formatted in the SPARQL query results JSON format[29], so that before presenting the results to the user they have to be formatted, which is handled by the `stringfyResultsJSON()` method.

```
/**
* Gets answer to query from qa system.
* @param {string} question to send to qa system
*/
async function askQuestion(question) {
    const requestConfig = {
        timeout: 50000,
        params: {
            question: question
        }
    };
    return new Promise((resolve, reject) => {
        axios.get(qaURL, requestConfig)
            .then(res => {
                res.data.askQuery = 'boolean' in res.data.answer;
                res.data.answer = stringfyResultsJSON(res.data.answer);
                resolve(res.data);
            })
            .catch(err => {
                if (err.response && err.response.status == 500) {
                    reject('noanswer');
                } else {
                    reject('other');
                }
            })
    });
}
```

Listing 4.5: Sending a question to the QA system via HTTP GET request.

[28] https://www.npmjs.com/package/axios
[29] https://www.w3.org/TR/sparql11-results-json/

4.3.5 Implemented Conversation Flows

The bot module currently supports two main a greeting-and-introduction and a question-and-answer conversation flow.

The greeting conversation is triggered when the user sends one of the following messages: “hi”, “hello”, “howdy”, “hey”, “aloha”, “hola”, “bonjour”, “oi”, “hallo”, “moin”. If triggered, the bot answers randomly with one of the texts seen in Listing 4.6 and also includes the quick reply options “opal?”, “open data?”, “themes?” and “for example?”. Figure 4.7 and 4.8 show how these quick reply options are displayed in the web- and Twitter UI. Those, when clicked, trigger answers from the bot that either explain the terms in the case of “opal” and “open data” or list all themes for which data exists in the database or in the case of the “for example?” quick reply provide an example question and its answer from the QA system. Figure 4.5 shows an example of an introductory conversation in the web UI and Figure 4.3 the same one in the Twitter direct message UI.

The question and answer dialog is executed whenever none of the other conversations are set off. Figure 4.6 shows the flow of activities of the bot in a question-and-answer conversation. There are four different scenarios and three different answers the bot gives. The HTTP error 500 is used by the QA module to signal that it could not answer the question. When the bot receives it, it tells the user that their question could not be answered and asks if they have another one. If the request fails with a different error or if the request times out and no answer was received at all, the bot responds that the QA system is unavailable and the user should try again later. If everything works correctly and the QA module sends a valid answer to the question, the bot first sends the answer and then asks if the user either wants to see more results for the current question or has other questions. For a QA conversation via tweets, the flow differs as the bot only replies to the question with the results and provides a link to the Fuseki instance where the query the QA system used to answer the question is prefilled and more results can be fetched. Figure 4.9 shows an example of a QA conversation via tweets.

```
1  const greetings = [
2      'Hello, I am the OPAL open data bot! You can ask me questions about
       ↪  the metadata in the OPAL database.',
3      'Hi! I am the OPAL open data bot and answer question about the
       ↪  metadata in the OPAL database.',
4      'Hey! The OPAL open data bot here. I answer question about the
       ↪  metadata in the OPAL database.'
5  ];
```

Listing 4.6: Greeting variants.

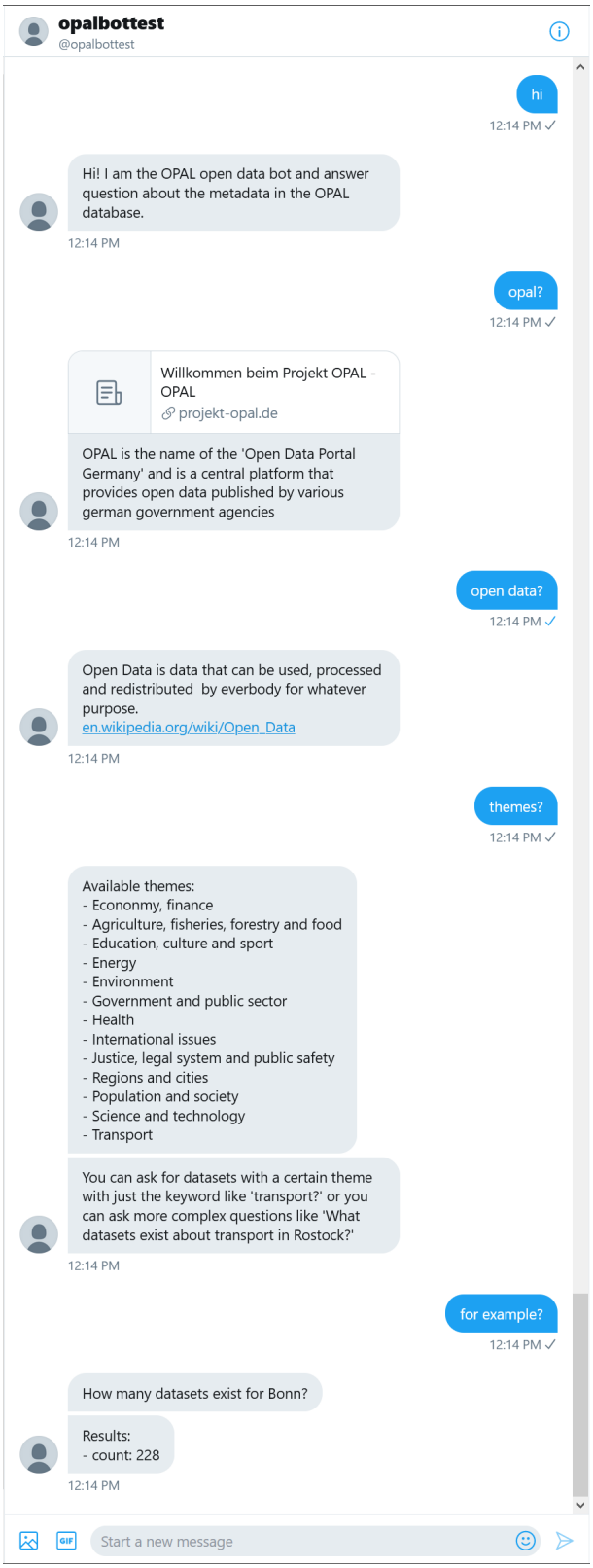

Figure 4.3: Twitter direct message UI.

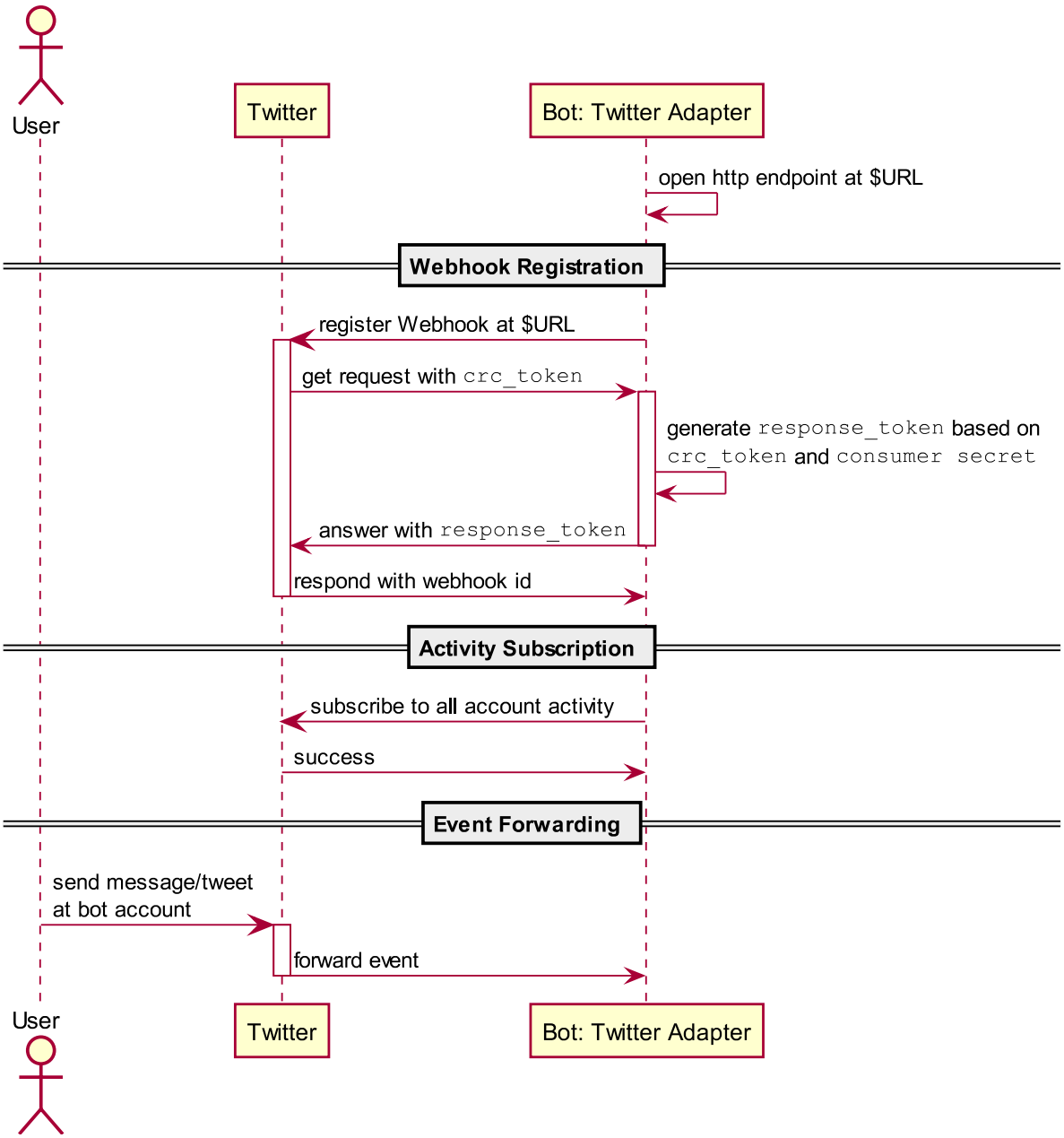

Figure 4.4: Successful Twitter adapter webhook registration.

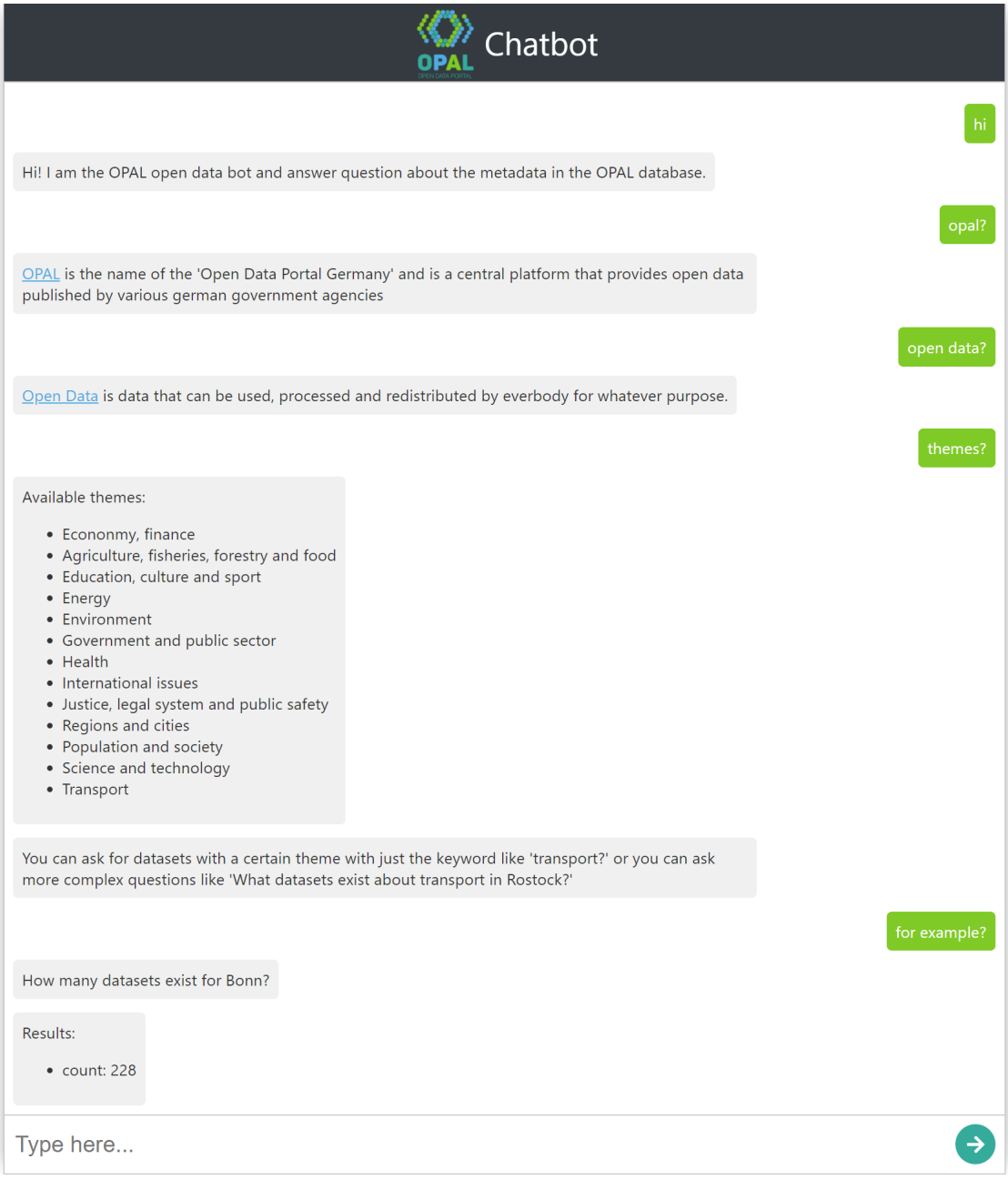

Figure 4.5: Botkit web UI with messages.

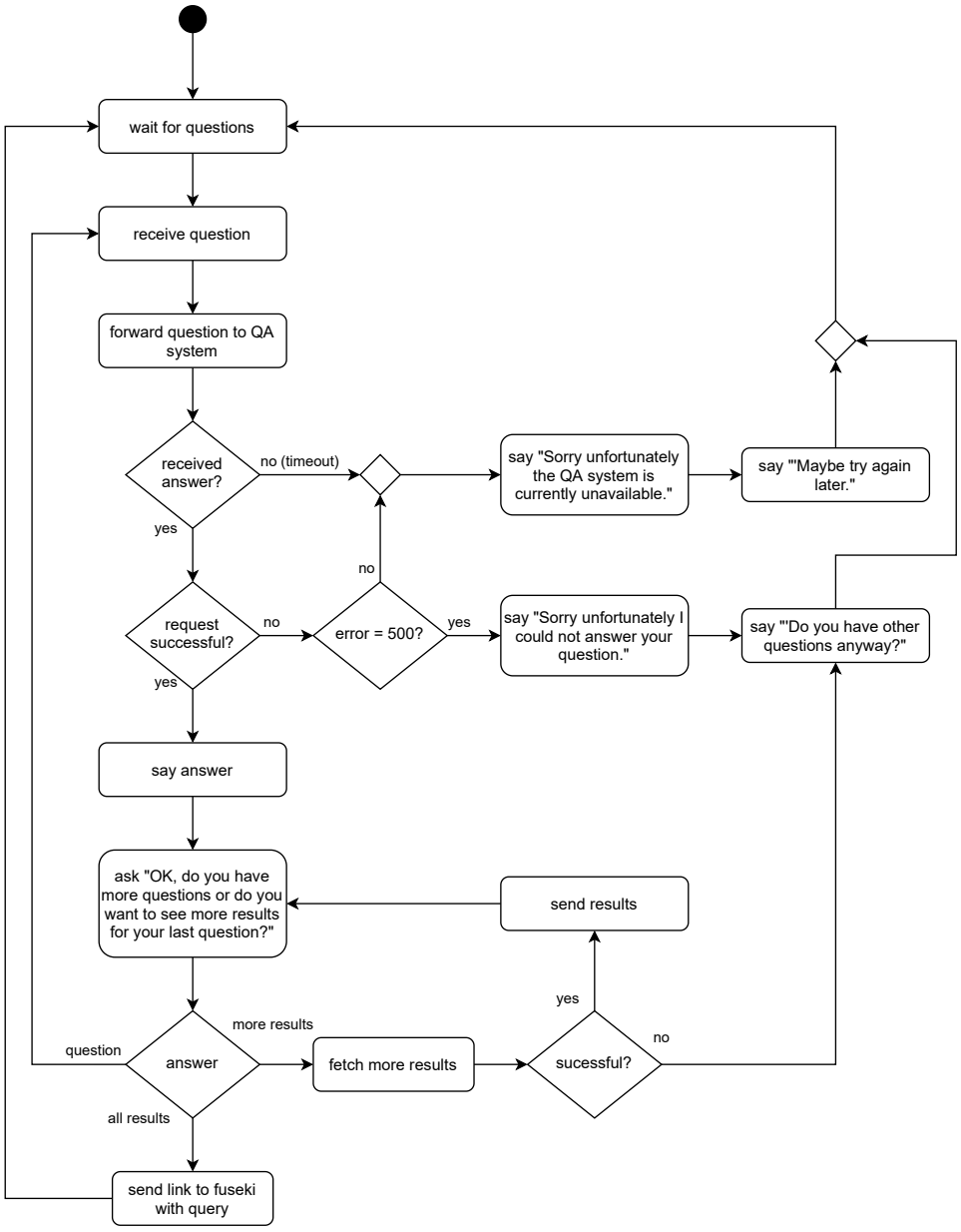

Figure 4.6: Activity flow of the bot in a QA dialog.

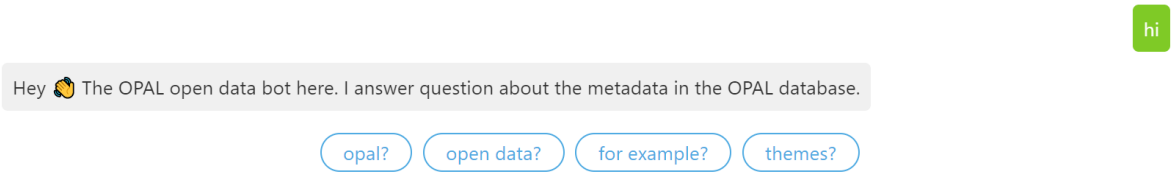

Figure 4.7: Quick reply options in the web UI.

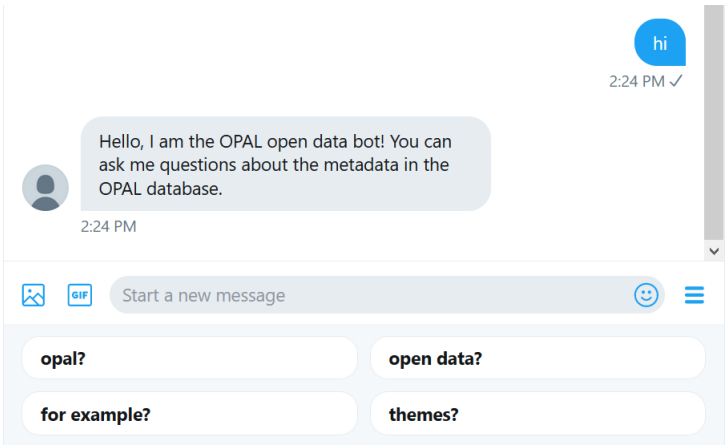

Figure 4.8: Quick reply options in the Twitter UI.

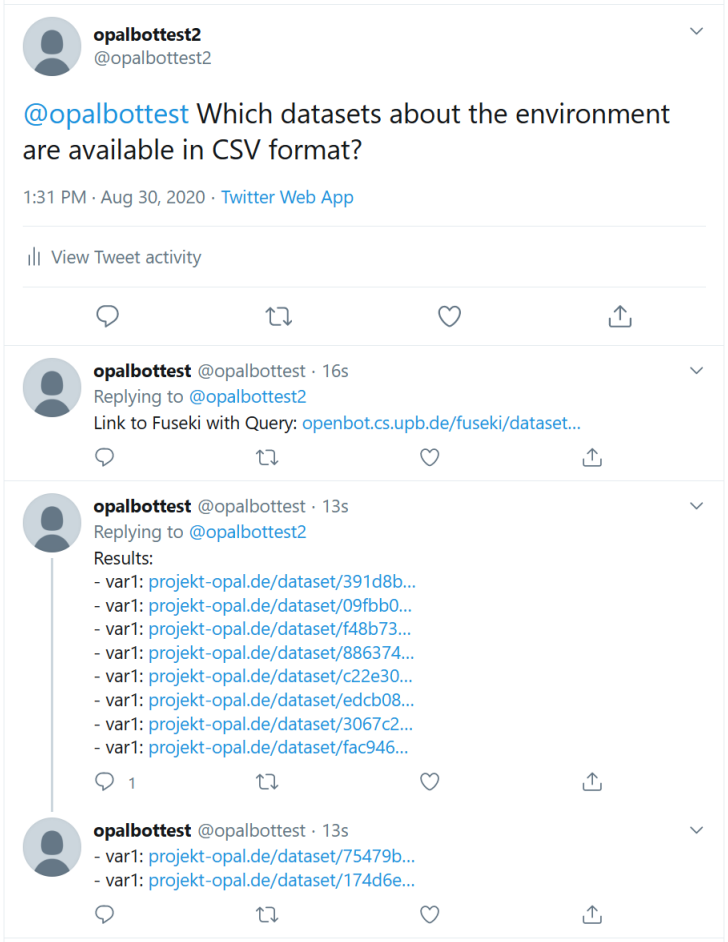

Figure 4.9: QA dialog via tweets.

5 Evaluation

To measure the capabilities and discover potential remaining weaknesses, the QA module was evaluated on a dataset of 50 questions annotated with the optimal SPARQL query and its results. In the following, the process of the creation and annotation of the question set as well as the final evaluation will be explained. Figure 5.1 gives an overview of the steps involved in the evaluation procedure.

5.1 Dataset

The first step in the evaluation process was the creation of a set of questions. As mentioned before in Section 3.4.2, the questions had to be created manually because there are no existing datasets for neither the DCAT vocabulary nor the OPAL database. The goal during their creation was to have a set of questions that covers each DCAT property and includes simple as well as more complex questions. More specifically, the following types of questions were included:

- questions asking for all values of a DCAT property
- questions with language, frequency, theme, license, location or file type entities
- questions containing time expressions
- questions containing string literals
- questions asking for the number of datasets or distributions, catalogs, etc.
- questions asking for the superlatives of certain DCAT properties

All questions were composed in German but later translated to English, as described in the following section.

5.2 Annotation and Annotator Agreement

To employ the created question set for benchmarking the QA system, the questions have to be annotated with the SPARQL queries that deliver the optimal answers to the questions. To remove bias and consider a second viewpoint from someone not involved in the development of the system, this was done not only by me but also by a second independent annotator. For

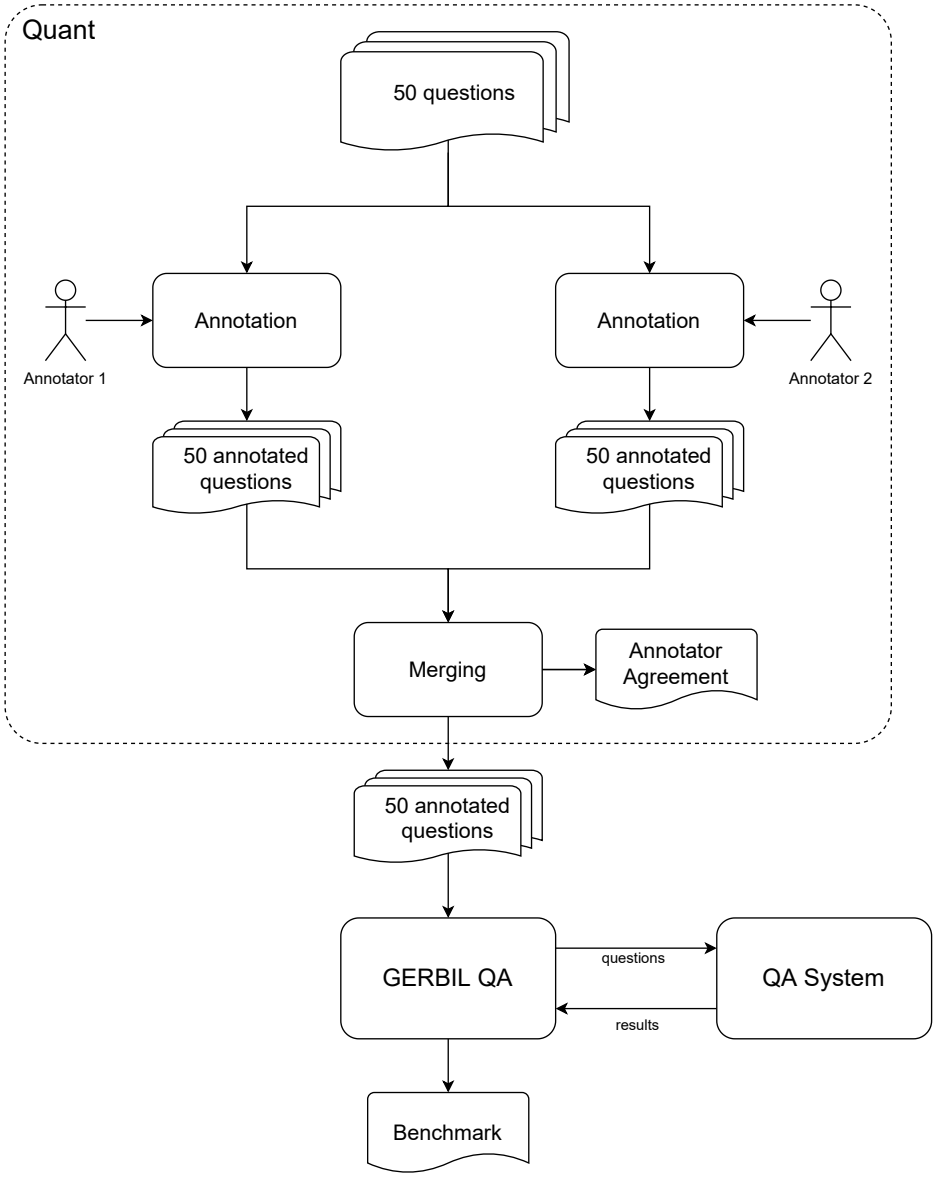

Figure 5.1: Overview of the evaluation process.

the annotation and later merging of annotations the benchmark curation QUANT[Gus+19] was used. QUANT is a framework for the creation and curation of QA benchmarks and alleviates the manual alteration of the benchmarks. It offers smart edit suggestions as well as quality checks for queries. As can be seen in Figure 5.1, QUANT was utilized for the creation of the question set, the annotation, and the merging of the annotations. In particular, the annotations added to the questions include:

- the expected answer type (resource, number, string, ...)
- if the query includes aggregation functions (COUNT, MAX, MIN, ...)
- if the query contains URIs outside DBpedia, which is true for all questions and, thus, can be ignored
- if the questions requires searching both linked data and textual data, which is false for all questions and, thus, can be ignored

- the keywords of the German question sentence
- the English translation of the question sentence including the keywords
- the optimal SPARQL query delivering the answers to the question
- the expected results of the query
- the actual results of the SPARQL query, which are fetched automatically by QUANT from the database endpoint

After both annotators completed the annotations of all 50 questions with the just mentioned properties, the annotations could then be combined to a final set of annotated questions which served as basis for the benchmark. The merging involved both annotators going over all questions, discussing the differences of each of the annotations, and at the end agreeing on one of the two annotations for each question. Because many of the questions were annotated very similar by both annotators, there was no major discord during the merging process and a third independent annotator was not needed. Minor points of discussion were:

- How queries for questions like "What are the latest datasets?" are limited. Meaning how many results should be considered the latest. The limit agreed upon was 10.
- If and how queries that frequently time out because there are so many results, should be limited. The limit agreed upon was 100 results.

The impact of the unification of the limits can be seen when comparing the annotator agreement before and after the unification, cf. Section 5.2.3.

5.2.1 Setup

The annotator agreements were calculated with GERBIL QA [Usb+19]. GERBIL QA was developed by Vollmers et al. and is a benchmarking platform for question and answering systems based on the GERBIL framework for benchmarking Named Entity Recognition and Entity Linking systems. GERBIL QA was also used for the benchmarking of the QA system. The annotator agreements were calculated by exporting both of the annotated questions sets from QUANT and configuring GERBIL experiments where one set serves as benchmark dataset and the other as answer dataset.

5.2.2 Evaluation Metrics

GERBIL QA outputs 9 different evaluation metrics, including micro- and macro precision, recall and F1 measure, a QALD specific F1 measure, the runtime and the number of errors for webservice-based QA systems. The two latter ones can be disregarded for the annotator agreement.

In the following, Q is the set of 50 questions, $a_q \in A$ are the results for the question $q \in Q$ from the annotator, $d_q \in D$ are the results for the question from the benchmark dataset, and $M(x, y)$ specifies a match of two entities. With the following definitions of true/false positives and true/false negatives:

$$
\begin{aligned}
tp(a_q, d_q, M) &= \{x \in a_q \mid \exists x' \in d_q : M(x, x')\} \\
fp(a_q, d_q, M) &= \{x \in a_q \mid \nexists x' \in d_q : M(x, x')\} \\
tn(a_q, d_q, M) &= \{x \notin a_q \mid \nexists x' \in d_q : M(x, x')\} \\
fn(a_q, d_q, M) &= \{x \in d_q \mid \nexists x' \in a_q : M(x, x')\}
\end{aligned}
$$

Annotator	Dataset	Micro F1	Micro Precision	Micro Recall	Macro F1	Macro Precision	Marco Recall	Macro F1 QALD	Answertype
Annot. A	Annot. B	0.4114	0.3216	0.5707	0.8035	0.8403	0.9227	0.8796	0.88
Annot. B	Annot. A	0.4114	0.5707	0.3216	0.8035	0.9227	0.8403	0.8796	0.88

Table 5.1: Annotator agreement before unifying limits.

and the definitions of precision, recall and the F1 measure:

$$P(a_q, d_q, M) = \frac{|tp(a_q, d_q, M)|}{(|tp(a_q, d_q, M)| + |fp(a_q, d_q, M)|)} \quad \text{(Precision)}$$

$$R(a_q, d_q, M) = \frac{|tp(a_q, d_q, M)|}{(|tp(a_q, d_q, M)| + |fn(a_q, d_q, M)|)} \quad \text{(Recall)}$$

$$F1(a_q, d_q, M) = \frac{2 \cdot P(a_q, d_q, M) \cdot R(a_q, d_q, M)}{P(a_q, d_q, M) + R(a_q, d_q, M)} \quad \text{(F1)}$$

the metrics outputted by GERBIL are defined as:

$$P_{mic}(A, D, M) = \frac{\sum_{q \in Q} |tp(a_q, d_q, M)|}{\sum_{q \in Q} (|tp(a_q, d_q, M)| + |fp(a_q, d_q, M)|)} \quad \text{(Micro Precision)}$$

$$R_{mic}(A, D, M) = \frac{\sum_{q \in Q} |tp(a_q, d_q, M)|}{\sum_{q \in Q} (|tp(a_q, d_q, M)| + |fn(a_q, d_q, M)|)} \quad \text{(Micro Recall)}$$

$$F1_{mic}(A, D, M) = \frac{2 \cdot P_{mic}(A, D, M) \cdot R_{mic}(A, D, M)}{P_{mic}(A, D, M) + R_{mic}(A, D, M)} \quad \text{(Micro F1)}$$

$$P_{mac}(A, D, M) = \frac{\sum_{q \in Q} P(a_q, d_q, M)}{|Q|} \quad \text{(Macro Precision)}$$

$$R_{mac}(A, D, M) = \frac{\sum_{q \in Q} R(a_q, d_q, M)}{|Q|} \quad \text{(Macro Recall)}$$

$$F1_{mac}(A, D, M) = \frac{2 \cdot P_{mac}(A, D, M) \cdot R_{mac}(A, D, M)}{P_{mac}(A, D, M) + R_{mac}(A, D, M)} \quad \text{(Macro F1)}$$

The macro F1 QALD metric is included for comparability with older QALD challenges. It differs from the macro F1 metric as it scores empty answers from the annotator differently. The details about the additional semantic information used for the scores can be found in the GERBIL QA paper [Usb+19]. As can be seen in the definitions above, the macro scores are the average over the corresponding scores of each question and the micro scores consider all annotations together and therefore put more weight on questions with more results.

Annotator	Dataset	Micro F1	Micro Precision	Micro Recall	Macro F1	Macro Precision	Marco Recall	Macro F1 QALD	Answertype
Annot. A	Annot. B	0.8482	0.9422	0.7713	0.9451	0.9337	0.988	0.9601	0.88
Annot. B	Annot. A	0.8482	0.7713	0.9422	0.9451	0.988	0.9337	0.9601	0.88

Table 5.2: Annotator agreement after unifying limits.

5.2.3 Interpretation of the Results

Table 5.1 and Table 5.2 show both combinations of benchmark and answer datasets, which results in a reversal of the recall and precision scores. The macro F1 measure of 0.988, which weights every question the same, shows that both annotations where very similar and the annotators almost agreed on all the questions.

The micro F1 measure more than the doubled in comparison to before adjusting the limits. This makes sense as the adjusting of the limits especially effects the queries with many results. The macro F1 measure increases as well with the unified limits, but not as significantly.

The annotated answer types coincided for 88% of the questions, so for 44 out of 50. When taking into account the probability of the annotation of the answer types coinciding by chance, by calculating the Cohen's kappa coefficient, the inter-rater reliability is 73.4%. During the merging discussion it was, however, revealed that the only times the answer types did not coincide were when one of the queries was faulty. For example, the question "What is the most used license for German datasets?" requires a query with a `GROUP BY`, an `ORDER BY` and a `COUNT` modifier and was annotated by one annotator with a query that selected the license as well as the count so the answer type was neither only a number nor only a resource. The correct query would combine the `ORDER BY` and a `COUNT` modifiers and only select the license variable.

So generally, the annotators agreed in the majority of the cases and, therefore, it can be concluded that the questions are unambiguous in the sense that they can be represented by and answered with a distinct query.

5.3 Benchmark

5.3.1 Setup

As Figure 5.1 shows, the merged question set was used for benchmarking the QA system with GERBIL QA in the next step. For this, two new GERBIL experiments were configured. One for the German and one for the English questions. Both with the merged question set as dataset and the QA system endpoint as annotator. The QA system ran together with the Elasticsearch index and the Apache Jena Triple Store on a virtual machine with 2 cores of an `Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz` and 20 gigabytes of RAM. All three applications ran inside their own Docker container. The results can be seen in Table 5.3.

Language	Micro F1	Micro Precision	Mirco Recall	Macro F1	Macro Precision	Marco Recall	Macro F1 QALD	Average Answertime
en	0.0727	0.0379	0.8563	0.5876	0.63	0.7487	0.7872	5.844s
de	0.1538	0.0854	0.7753	0.4759	0.5086	0.62	0.7165	5.588s

Table 5.3: Benchmark Results

5.3.2 Interpretation of the Results

The micro precision results stand out from the other measures as very low. This is a result of the limits of the queries in the question set, which were agreed on during the merging, cf. Section 5.2. The QA system does not add limits to the queries and waits up to ten seconds for results from the Fuseki endpoint in its current configuration. This has the consequence that for questions like "What data formats are available?" the question set only contains 100 results whereas the QA system answers with more than 80.000. As mentioned before, the micro measures put more weight on questions with many results which leads to a few questions being responsible for the low scores.

Overall, the macro F1 score and macro F1 QALD score show that the QA system is able to confidently answer the majority of the questions. The benchmark demonstrates that the system can answer simple questions like "Which data formats are available?" and "What datasets exist for Rostock?" as well as more complex questions like "What are download urls of distributions published this year?". However, questions like "What is the frequency with which most datasets about transport are updated?", that require very complex queries containing a `GROUP BY` as well as a `COUNT` and `ORDER BY` modifier, currently can not be answered by the QA system. The problem with questions like these is, that there are no indicators that a `GROUP BY` or `COUNT` modifier is necessary to build a fitting query and thus even if there exist templates with these modifiers the QA system does not know when to use them.

Furthermore, the QA system is currently not able to handle English names of cities like "Munich" for "München" as the labels for the location entities only include the German names. That is why the QA system can not answer the question "What is there about the topic culture in Munich?" correctly but can answer the German translation "Was gibt es zum Thema Kultur in München?". However, this is due to the missing data and not the QA mechanism.

Another question from the dataset that could not be answered and points out a weakness of the QA system is "Which datasets are available as PDF as well as JSON?". To answer this question the system would have to recognize that a dataset with two different distributions is asked for and, thus, the query has to include the property `dcat:distribution` two times pointing to distributions having a `dct:format` property. Currently, the system only recognizes the `dcat:dataset` and the file type entities. Also, there is no template for a query structure like the one needed in this case.

Table 5.3 also lists the average time the QA system needed to answer the questions (en: 5.844s, de: 5.588s). In comparison to other systems like TeBaQA (2.553s)[Vol+] and wdaqua-core1 (0.446s and 0.436s) [DSM18] this is quite high. The part that takes the longest during the QA process is the query execution. The system considers multiple SPARQL queries for each

question before picking a final result and thus queries, that take the Fuseki instance a long time to execute, slow down the QA process. As mentioned before the configured timeout for Fuseki queries is 10 seconds.

In conclusion, the benchmark demonstrates the fundamental capabilities of the QA system but does not provide a basis for a performance comparison with other systems. As there currently do not exist any other systems for DCAT and the OPAL database, a comparison is not possible anyway, but for the development of future systems an independent benchmark dataset should be created so that the systems can be evaluated in a comparable manner.

6 Future Work

Throughout this thesis, many opportunities for the extension and improvement of the presented approach were already broached. One of them is the possible expansion of the bot to other messaging platforms to reach a larger audience. Jalota et al. identified in their analysis of the DBpedia Chatbot[Jal+20] multiple features as beneficial for conversational interfaces, that are currently not implemented by the bot module. They include a user input guidance with auto-completion and auto-correction, a system that explains what went wrong if a question could not be answered, and the detection of out-of-scope questions and a following explanation of the capabilities of the system. These could be added to the bot module as well.

The QA system could also be extended in multiple ways. As mentioned in Section 3.5.4, it was designed to be easily extensible with support for more languages, so expanding the multilingualism is one way. Another goal of future work could also be the improvement of the core QA capabilities, for example, by adding support for more complex queries or adding English labels for location entities as described during the evaluation, cf. Section 5.3. Furthermore, the templates could be simplified by removing modifiers like `ORDER BY` and adding them during the query building process. This would lead to fewer templates as only one is needed for a specific triple structure and not an extra one for each modifier.

7 Summary and Conclusion

This thesis proposed a Social Media chatbot in combination with a QA system for the DCAT vocabulary and the open data platform OPAL. After the introduction of the main concepts in the first chapter, some related articles and papers, that are already published and cover the same topics as this thesis, were presented. In the third chapter the implemented QA system was presented. First the scenario and requirements were established and an overview of existing QA approaches was given. The analysis of the data and review of the techniques of other systems led to the decision to implement an own approach instead of adapting an existing one. Subsequently, the details of the implementation were elucidated. In conclusion, the presented system does not realise a completely new strategy to solve the QA problem but does connect proven methods to fulfill the requirements of the bot application and, thus, reaches the goal of this thesis. However, the recognition of time entities and string literals is, to the best of my knowledge, not supported by any existing QA systems and, thus, presents a novel method of QA. Furthermore, the application of the implemented QA functionalities to the DCAT vocabulary is novel.

The fourth chapter of the thesis revolved around the conversational interface for the QA system, i.e. the bot module. The requirement was, that the bot module should be connected to an existing messaging platform, so, first, different platforms were presented as candidates. Then the decision process for one platform was explained and after that further requirements were established and the development of the bot module was detailed.

In the end the QA approach was evaluated with a newly created and annotated benchmark dataset. It was demonstrated that the QA system is capable of answering simple as well as more complex questions but fails to handle questions requiring complex query structures. Chapter six addressed the mentioned weaknesses and pointed out ways to further improve the bot as well as the QA system.

The code for the QA system and the bot modules is open source and can be found on Github[1].

[1] `https://github.com/martenls/dcat-qa-system`

Bibliography

[Alb+20] Riccardo Albertoni et al. *Data Catalog Vocabulary (DCAT) - Version 2*. Feb. 4, 2020. URL: https://www.w3.org/TR/vocab-dcat-2/ (visited on 04/17/2020).

[ANU18] Ram G. Athreya, Axel-Cyrille Ngonga Ngomo, and Ricardo Usbeck. "Enhancing Community Interactions with Data-Driven Chatbots–The DBpedia Chatbot". In: *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*. Companion of the The Web Conference 2018. Lyon, France: ACM Press, 2018, pp. 143–146. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3186964. URL: http://dl.acm.org/citation.cfm?doid=3184558.3186964 (visited on 07/15/2019).

[BG20] Hadas Bitra and Jean Gabarra. *Delivering Information and Eliminating Bottlenecks with CDC's COVID-19 Assessment Bot*. Mar. 20, 2020. URL: https://blogs.microsoft.com/blog/2020/03/20/delivering-information-and-eliminating-bottlenecks-with-cdcs-covid-19-assessment-bot/ (visited on 09/10/2020).

[Bor+15] Antoine Bordes et al. "Large-Scale Simple Question Answering with Memory Networks". In: (June 5, 2015). arXiv: 1506.02075 [cs]. URL: http://arxiv.org/abs/1506.02075 (visited on 07/25/2020).

[Bot+16] Andreas Both et al. "Qanary – A Methodology for Vocabulary-Driven Open Question Answering Systems". In: *The Semantic Web. Latest Advances and New Domains*. Ed. by Harald Sack et al. Vol. 9678. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 625–641. ISBN: 978-3-319-34128-6. DOI: 10.1007/978-3-319-34129-3_38. URL: http://link.springer.com/10.1007/978-3-319-34129-3_38 (visited on 03/19/2020).

[CM12] Angel X. Chang and Christopher D. Manning. "SUTime: A Library for Recognizing and Normalizing Time Expressions". In: *Proceedings of the Eighth International Conference on Language Resources and Evaluation, LREC 2012, Istanbul, Turkey, May 23-25, 2012*. 2012, pp. 3735–3740. URL: http://www.lrec-conf.org/proceedings/lrec2012/summaries/284.html.

[Dal16] Robert Dale. "The Return of the Chatbots". In: *Natural Language Engineering* 22.5 (2016), pp. 811–817. DOI: 10.1017/S1351324916000243. URL: https://doi.org/10.1017/S1351324916000243.

[DSM18] Dennis Diefenbach, Kamal Singh, and Pierre Maret. "WDAqua-Core1: A Question Answering Service for RDF Knowledge Bases". In: *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*. Companion of the The Web Conference 2018. Lyon, France: ACM Press, 2018, pp. 1087–1091. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3191541. URL: http://dl.acm.org/citation.cfm?doid=3184558.3191541 (visited on 07/25/2020).

[Fou] Open Knowledge Foundation. *The Open Definition - Open Definition - Defining Open in Open Data, Open Content and Open Knowledge*. URL: http://opendefinition.org/ (visited on 04/15/2020).

[Gro14] Government Linked Data (GLD) Working Group. *Data Catalog Vocabulary (DCAT)*. Jan. 16, 2014. URL: https://www.w3.org/TR/2014/REC-vocab-dcat-20140116/ (visited on 09/09/2020).

[Gus+19] Ria Hari Gusmita et al. "QUANT - Question Answering Benchmark Curator". In: *Semantic Systems. The Power of AI and Knowledge Graphs*. Ed. by Maribel Acosta et al. Vol. 11702. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 343–358. ISBN: 978-3-030-33219-8 978-3-030-33220-4. DOI: 10.1007/978-3-030-33220-4_25. URL: http://link.springer.com/10.1007/978-3-030-33220-4_25 (visited on 07/29/2020).

[Hu+18] Sen Hu et al. "Answering Natural Language Questions by Subgraph Matching over Knowledge Graphs". In: *IEEE Transactions on Knowledge and Data Engineering* 30.5 (May 2018), pp. 824–837. ISSN: 1558-2191. DOI: 10.1109/TKDE.2017.2766634.

[JAB19] Thorhildur Jetzek, Michel Avital, and Niels Bjørn-Andersen. "The Sustainable Value of Open Government Data". In: *J. AIS* 20.6 (2019), p. 6. URL: https://aisel.aisnet.org/jais/vol20/iss6/6 (visited on 11/27/2019).

[Jal+20] Rricha Jalota et al. "An Approach for Ex-Post-Facto Analysis of Knowledge Graph-Driven Chatbots – The DBpedia Chatbot". In: *Chatbot Research and Design*. Ed. by Asbjørn Følstad et al. Vol. 11970. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 19–33. ISBN: 978-3-030-39539-1. DOI: 10.1007/978-3-030-39540-7_2. URL: http://link.springer.com/10.1007/978-3-030-39540-7_2 (visited on 05/02/2020).

[KSV19] Sophia Keyner, Vadim Savenkov, and Svitlana Vakulenko. "Open Data Chatbot". In: *The Semantic Web: ESWC 2019 Satellite Events - ESWC 2019 Satellite Events, Portorož, Slovenia, June 2-6, 2019, Revised Selected Papers*. Ed. by Pascal Hitzler et al. Vol. 11762. Lecture Notes in Computer Science. Springer, 2019, pp. 111–115. ISBN: 978-3-030-32326-4. DOI: 10.1007/978-3-030-32327-1_22.

[LK19] Andreas Lommatzsch and Jonas Katins. "An Information Retrieval-Based Approach for Building Intuitive Chatbots for Large Knowledge Bases". In: *Proceedings of the Conference on "Lernen, Wissen, Daten, Analysen", Berlin, Germany, September 30 - October 2, 2019*. Ed. by Robert Jäschke and Matthias Weidlich. Vol. 2454. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 343–352. URL: http://ceur-ws.org/Vol-2454/paper_60.pdf (visited on 11/23/2019).

[Luk+17] Denis Lukovnikov et al. "Neural Network-Based Question Answering over Knowledge Graphs on Word and Character Level". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17: 26th International World Wide Web Conference. Perth Australia: International World Wide Web Conferences Steering Committee, Apr. 3, 2017, pp. 1211–1220. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052675. URL: https://dl.acm.org/doi/10.1145/3038912.3052675 (visited on 07/25/2020).

[MKS18] Ines Mergel, Alexander Kleibrink, and Jens Sörvik. "Open Data Outcomes: U.S. Cities between Product and Process Innovation". In: *Government Information Quarterly* 35.4 (Oct. 1, 2018), pp. 622–632. ISSN: 0740-624X. DOI: 10.1016/j.giq.2018.09.004. URL: http://www.sciencedirect.com/science/article/pii/S0740624X18300017 (visited on 02/09/2020).

[Rus+15] Stefan Ruseti et al. “QAnswer - Enhanced Entity Matching for Question Answering over Linked Data”. In: *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation Forum, Toulouse, France, September 8-11, 2015*. Ed. by Linda Cappellato et al. Vol. 1391. CEUR Workshop Proceedings. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1391/99-CR.pdf.

[Sta] Statista. *Most Popular Messaging Apps 2019*. URL: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/ (visited on 03/22/2020).

[Ung+12] Christina Unger et al. “Template-Based Question Answering over RDF Data”. In: *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web* (Apr. 16, 2012). DOI: 10.1145/2187836.2187923.

[Ung+15] Christina Unger et al. “Question Answering over Linked Data (QALD-5)”. In: *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation Forum, Toulouse, France, September 8-11, 2015*. 2015. URL: http://ceur-ws.org/Vol-1391/173-CR.pdf.

[Usb+18] Ricardo Usbeck et al. “8th Challenge on Question Answering over Linked Data (QALD-8) (Invited Paper)”. In: *Joint Proceedings of the 4th Workshop on Semantic Deep Learning (SemDeep-4) and NLIWoD4: Natural Language Interfaces for the Web of Data (NLIWOD-4) and 9th Question Answering over Linked Data Challenge (QALD-9) Co-Located with 17th International Semantic Web Conference (ISWC 2018), Monterey, California, United States of America, October 8th - 9th, 2018*. 2018, pp. 51–57. URL: http://ceur-ws.org/Vol-2241/paper-05.pdf.

[Usb+19] Ricardo Usbeck et al. “Benchmarking Question Answering Systems”. In: *Semantic Web* 10.2 (Jan. 21, 2019). Ed. by Ruben Verborgh et al., pp. 293–304. ISSN: 22104968, 15700844. DOI: 10.3233/SW-180312. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-180312 (visited on 08/27/2020).

[Vol+] Daniel Vollmers et al. “TeBaQA - Template-Based Question Answering Using Graph-Pattern Isomorphism”. Is not yet published.

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Eidesstattliche Versicherung

Nachname Schmidt Vorname Marten

Matrikelnr. 7090644 Studiengang Informatik Bachelor v4

(•) Bachelorarbeit ( ) Masterarbeit

Titel der Arbeit A Question Answering (QA) System for the Data Catalog Vocabulary (DCAT)

☐ Die elektronische Fassung ist der Abschlussarbeit beigefügt.

☒ Die elektronische Fassung sende ich an die/den erste/n Prüfenden bzw. habe ich an die/den erste/n Prüfenden gesendet.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit (Ausarbeitung inkl. Tabellen, Zeichnungen, etc.) selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Abschlussarbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung entspricht der gedruckten und gebundenen Fassung.

Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist die Vizepräsidentin / der Vizepräsident für Wirtschafts- und Personalverwaltung der Universität Paderborn. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz NRW in der aktuellen Fassung).

Die Universität Paderborn wird ggf. eine elektronische Überprüfung der Abschlussarbeit durchführen, um eine Täuschung festzustellen.

Ich habe die oben genannten Belehrungen gelesen und verstanden und bestätige dieses mit meiner Unterschrift.

Ort Bad Lippspringe Datum 10.09.20

Unterschrift M. Schmidt

Datenschutzhinweis:

*Die o.g. Daten werden aufgrund der geltenden Prüfungsordnung (Paragraph zur Abschlussarbeit) i.V.m. § 63 Abs. 5 Hochschulgesetz NRW erhoben. Auf Grundlage der übermittelten Daten (Name, Vorname, Matrikelnummer, Studiengang, Art und Thema der Abschlussarbeit) wird bei Plagiaten bzw. Täuschung der*die Prüfende und der Prüfungsausschuss Ihres Studienganges über Konsequenzen gemäß Prüfungsordnung i.V.m. Hochschulgesetz NRW entscheiden. Die Daten werden nach Abschluss des Prüfungsverfahrens gelöscht. Eine Weiterleitung der Daten kann an die*den Prüfende*n und den Prüfungsausschuss erfolgen. Falls der Prüfungsausschuss entscheidet, eine Geldbuße zu verhängen, werden die Daten an die Vizepräsidentin für Wirtschafts- und Personalverwaltung weitergeleitet. Verantwortlich für die Verarbeitung im regulären Verfahren ist der Prüfungsausschuss Ihres Studiengangs der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.*