NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Geospatial Question Answering Web Application

Ioannis N. Maliaras

Supervisor: **Manolis Koubarakis,** Professor
Co-supervisor: **Dharmen Punjani,** Research Assistant

ATHENS

SEPTEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διαδικτυακή Εφαρμογή Γεωχωρικών Ερωτοαπαντήσεων

Ιωάννης Ν. Μαλιάρας

Επιβλέπων: **Μανώλης Κουμπαράκης,** Καθηγητής
Συνεπιβλέπων: **Dharmen Punjani,** Ερευνητικός Βοηθός

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2020

BSc THESIS

Geospatial Question Answering Web Application

Ioannis N. Maliaras
S.N.: 1115201500084

SUPERVISOR: **Manolis Koubarakis,** Professor
CO-SUPERVISOR: **Dharmen Punjani,** Research Assistant

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διαδικτυακή Εφαρμογή Γεωχωρικών Ερωτοαπαντήσεων

Ιωάννης Ν. Μαλιάρας
Α.Μ.: 1115201500084

ΕΠΙΒΛΕΠΩΝ: **Μανώλης Κουμπαράκης,** Καθηγητής
ΣΥΝΕΠΙΒΛΕΠΩΝ: **Dharmen Punjani,** Ερευνητικός Βοηθός

ABSTRACT

Question Answering over knowledge graphs has been studied a lot in recent years. The main aspect of such systems is to provide an interface, through which natural language questions can be posed and answered. Said systems generate queries and retrieve data from knowledge bases, usually in URI form. Thus, it is important to present this information appropriately, so that any user can make sense of the answers. We have developed an interface to the GeoQA system, which is a question answering engine over linked geospatial data. The interface of GeoQA is developed, having taken all different types of users in mind. By using this interface, a common user can pose a question in natural language and get the answer without knowing any of the underlying infrastructure. On the other hand, an expert user can analyze the QA engine and see the output of all the different modules. In addition, users can select different sets of data, over which they want to run the QA engine, as well as different components to complete different tasks. Therefore, we have developed an interface to realize the geospatial question answering engine GeoQA for all users, from the inquisitive scientist to the common layman.

SUBJECT AREA: Full-stack Web Development, User Interface Design, Natural Language Question Answering Systems

KEYWORDS: Knowledge Bases, SPARQL, Reactjs, Node.js, Docker

ΠΕΡΙΛΗΨΗ

Το πεδίο της απάντησης ερωτήσεων μέσω γράφων γνώσης έχει μελετηθεί πολύ τα τελευταία χρόνια. Η κύρια διάσταση τέτοιων συστημάτων, είναι η παροχή δεπαφής χρήστη, μέσω της οποίας καθίσταται δυνατή η θέση και η απάντηση ερωτήσεων σε φυσική γλώσσα. Τέτοια συστήματα παράγουν ερωτήματα και ανακτούν δεδομένα από βάσεις γνώσης, συνήθως σε μορφή URI. Έτσι λοιπόν, είναι σημαντικό να παρουσιάσουμε την πληροφορία αυτή κατάλληλα, έτσι ώστε να δύναται οποιοσδήποτε χρήστης να την κατανοήσει. Χτίσαμε λοιπόν, μια διεπαφή χρήστη για το σύστημα GeoQA, το οποίο είναι μια μηχανή ερωταπαντήσεων πάνω σε συνδεδεμένα γεωχωρικά δεδομένα. Η διεπαφή αυτή είναι σχεδιασμένη, λαμβάνοντας υπ' όψιν όλους τους τύπους χρηστών. Χρησιμοποιώντας τη διεπαφή, ένας απλός χρήστης μπορεί να θέσει μία ερώτηση σε φυσική γλώσσα και να λάβουν απάντηση χωρίς να γνωρίζουν τους εσωτερικούς μηχανισμούς. Από την άλλη μεριά, ένας ειδικευμένος χρήστης μπορεί να αναλύσει την μηχανή ερωταπαντήσεων και να ανακτήσει την έξοδο όλως των επιμέρους μονάδων. Επιπρόσθετα, οι χρήστες μπορούν να επιλέξουν διαφορετικά σύνολα δεδομένων, πάνω στα οποία επιθυμούν να εκτελέσουν, καθώς και διαφορετικές μονάδες του συστήματος για να επιτύχουν διαφορετικούς σκοπούς. Εν κατακλείδι, δημιουργήσαμε μια διεπαφή χρήστη που πραγματοποιεί την μηχανή GeoQA για όλων των ειδών χρήστες, από τον φιλέρευνο επιστήμονα, μέχρι τον μέσο, κοινό χρήστη.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Προγραμματισμός Ιστού Πλήρους Στοίβας, Σχεδιασμός Διεπαφής Χρήστη, Συστήματα Ερωταπαντήσεων σε Φυσική Γλώσσα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Γνωσιακές Βάσεις, SPARQL, Reactjs, Node.js, Docker

ACKNOWLEDGMENTS

I would like to wholeheartedly thank my supervisor Prof. Manolis Koubarakis, for allowing me to work with him, and use my expertise to help the research team achieve their goals.

Most of all, I would like to give my thanks to my advisor, and co-worker research assistant Dharmen Punjani, for his support, and dedication, despite any language barriers and hurdles we faced along the way.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

It’s not always easy for a computer science student to find a thesis subject. The field is vast and almost boundless. I struggled, at the start of this endeavor to find something that I could be passionate about. Fortune struck when I and my dear friend and classmate talked about possible subjects we wanted to work on. They said that there was this team, that worked under Professor Koubarakis, was looking to create a good user interface for their question answering system. A chance I seized in an instant. In my experience, not many people in the closed system of our department are interested in user-driven software or interface design, a field I’m very interested in since it can also be a creative outlet. I find it fascinating, being a middle man between the rectangular and deterministic software world, and the subconscious and impulsive human behavior, when dealing with said world. Thus, I sent the message to Dharmen to see what this was about.

1. INTRODUCTION

Recent times have seen the need for computer-aided natural language processing coming to the spotlight. Specifically, more and more systems that process, interpret, and answer questions in natural language e.g. “Which pubs are near the Guinness Brewery in Dublin?” are being implemented. Said systems are implemented on top of specialized databases called knowledge bases. Knowledge bases include DBpedia, Wikidata, Yago, and others.

The NKUA has built a system that answers more specific, **geospatial** questions, using a set of knowledge bases. That system is called GeoQA and it combines data from DBpedia, OpenStreetMaps, GADM, YAGO2, and YAGO2geo to answer questions such as:

- Where is Loch Ness located?
- Which bridges cross river Thames?

The purpose of this project is to create a user interface for the GeoQA system. This interface is designed in light of demonstrating and analyzing the system’s output in a user-friendly way, but also to potentially maintain and extend the system. The thesis takes the form of a full-stack web application, which collects all potential answers and displays them in a list and on a map, while also presenting the question’s technical and linguistic characteristics as they were interpreted by the system. This makes the system’s output easier to understand and analyze.

As it was implied, this application will be designed to be used by anyone, from the involved scientist/researcher and the system’s programmer, supervisor, or maintainer to the user with no background on knowledge bases and the such. It will provide a versatile demonstration tool while also acting as a helping hand to our research.

2. BASIC CONCEPTS

Before going into the specifics of our external development process, design choices, and external systems, we will provide some concise information about basic ideas regarding said specifics, without providing reasoning as of yet.

2.1 Web Application

By definition, *a web application is a computer program that utilizes web browsers and web technology to perform tasks over the internet.* A web application is a modular and abstracted computer program by nature. It is usually comprised of a web server, that manages requests by the user (client), an application server to perform whatever tasks the client asks it to perform, and sometimes, a database to store useful data.

There can be many ways to develop a web application. Some web applications are only static websites, that require no server-side processing. Other applications are much more dynamic, such as the one we have created for the purposes of this thesis, that require a multitude of modules, services both server and client-side.

Nowadays, web application development is divided into two stages. Back-end development and Front-end development. The term *Back-end* is referring to the application server. It is the things that the application does, without the user seeing it. For example, the Back-end queries the database, does resource-intensive computations, sends or shares files, and more. The Front-end on the other hand is what the user sees. It is the development of the user interface. Put very simply, it includes styling, design, formatting, and processing incoming data among others. Of course, both crafts have their theory, patterns, and gimmicks, and one could make a career out of each one specifically. There is, however, the field of **full-stack web development** where the developer creates the whole application, that is the back-end, the front-end, and whatever that implies. We're dealing with the development of a web application in full-stack.

The two parts of the web application communicate with each other and with other applications using an **Application Programming Interface (API).** An API acts as an intermediary software that makes communication between computer programs simple and friendly to developers. It adheres to standards and protocols, such as HTTP and REST, and it abstracts the underlying and dirty implementation of cross internet communication. In our case, any inter-software communication is achieved via APIs over the HTTP and/or REST protocols.

3. GEOSPATIAL QUESTION ANSWERING ENGINE: GEOQA

The GeoQA is a geospatial question answering system developed by D. Punjani et.al. [1]. It is implemented using reusable components as part of the Qanary [2] question answering methodology. The goal of this section is to introduce the reader to the system and make him familiar with the components that matter to us for this project. The basic structure of the system is shown below.

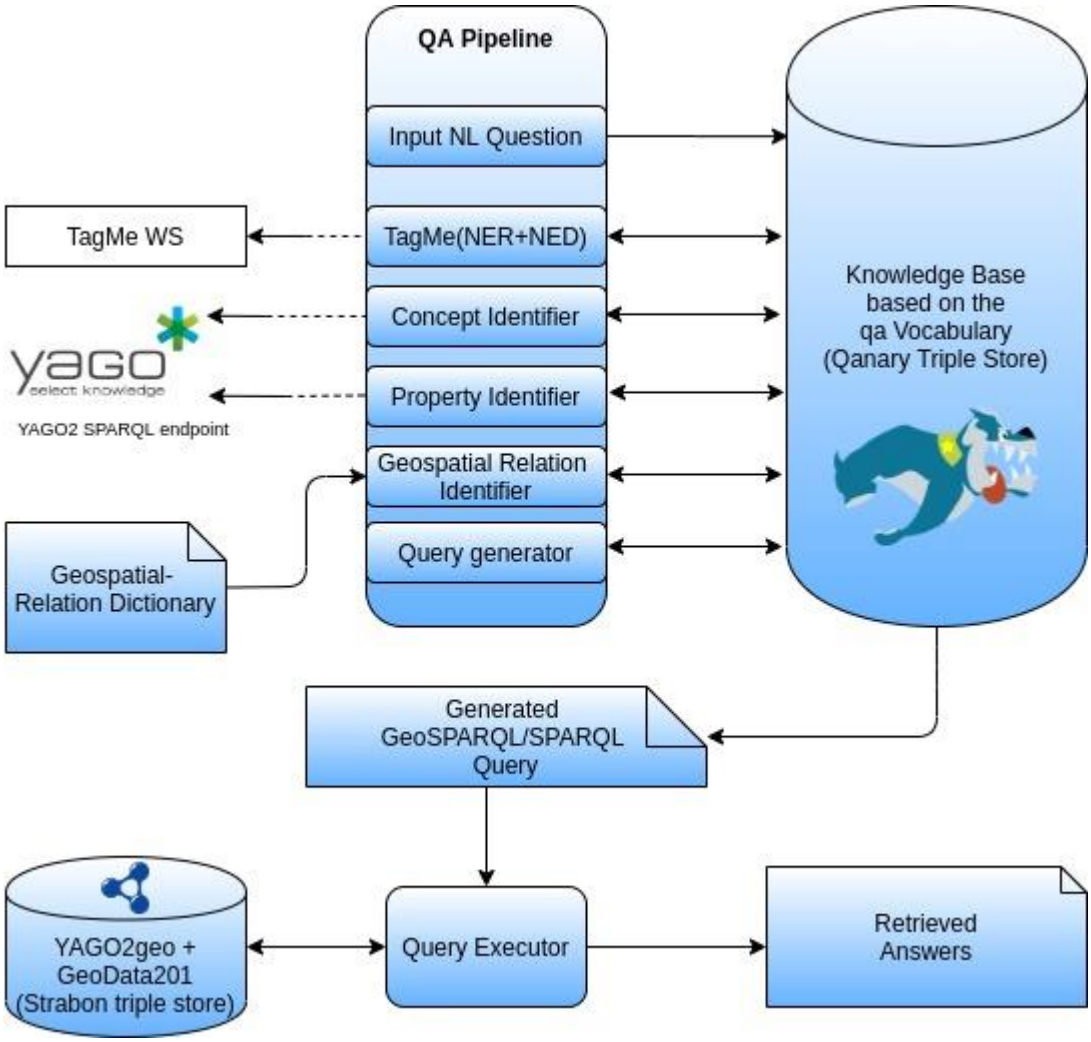

Figure 1: GeoQA System Basic Structure

We are mostly interested in the way that system retrieves and outputs information. We need to understand how to pose questions to the system and retrieve the answer and any other relevant information about the question analysis.

To be concise, the communication between the GeoQA system and the outside world is achieved via HTTP APIs that exist on every Qanary Pipeline Component as well as the Strabon Query Executor. We won't go into detail over how exactly those APIs expect input and return output, as it is not essential to the understanding of the project.

As for the second part of our exchange with the system, it would be helpful to understand what purpose each of the Qanary pipeline components has in the question analysis.

3.1 The Components

The Qanary pipeline components or **Components** as we will call them from now on are divided into 5 basic categories, each solving a different task for the question answering system.

1. Instance Identifier
2. Concept Identifier
3. Geospatial Relation Detector
4. Property Identifier
5. Query Generator

The components-part structure of the GeoQA system is shown below:

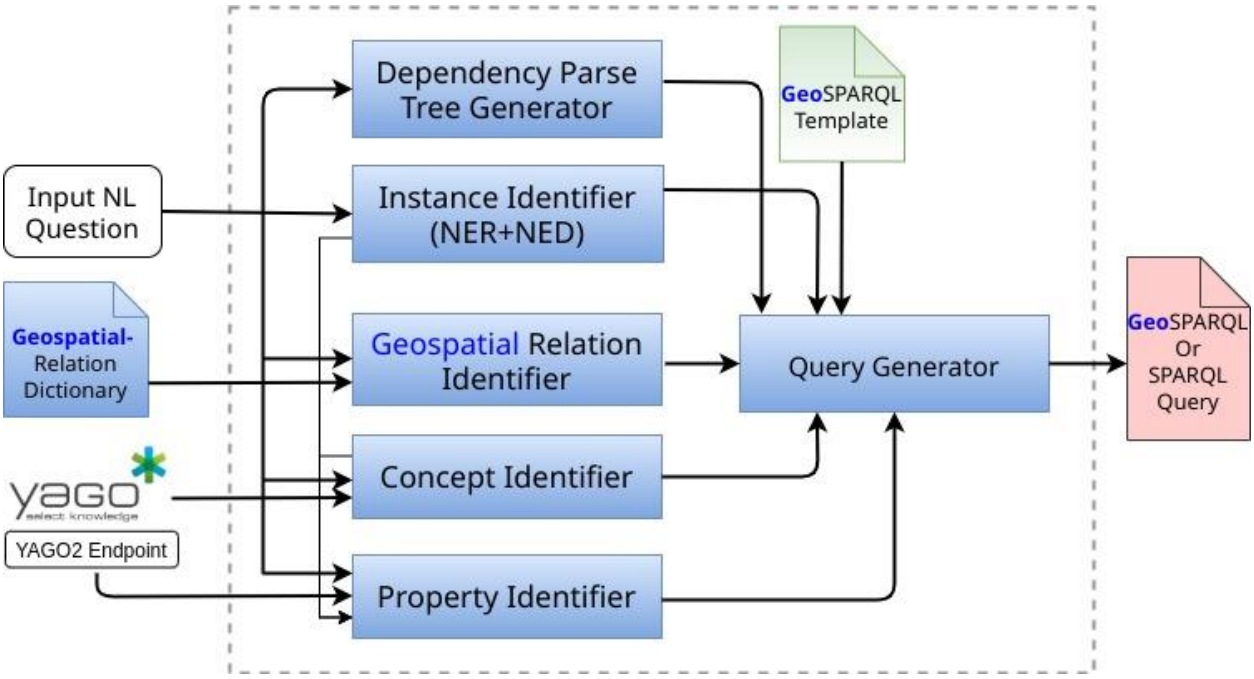

Figure 2: GeoQA System Components Structure

Let's briefly go over what each of the categories does.

3.1.1 Instance Identifier

The Instance Identifier's job consists of 2 phases:

- Named Entity Recognition (NER)
- Named Entity Disambiguation (NED)

During the NER phase, the component identifies named entities and classifies them. Entities can be anything from names of cities, rivers, people, to historical events, monetary values, and more. This is best understood using an example:

The NER subprocess takes this sentence as input:

- *Is Trafalgar Square located in London?*

And outputs:

- *Is (Trafalgar Square)$_{Location}$ located in (London)$_{Location}$?*

In the NED phase, the component links the named entities extracted to unique identities. In our specific example, the NED extension would link the 2 extracted locations to knowledge graphs. Specifically:

- Trafalgar Square à http://dbpedia.org/resource/Trafalgar_Square
- London à http://yago-knowledge.org/resource/London

It doesn't have to be DBpedia or Yago URIs specifically of course. Depending, on what dataset the system uses, the identities of the named entities change. We will be able to choose what dataset or dataset combinations we are using for each question, as we will discuss in later sections. From now on we will be talking about those entities as **Instances**

3.1.2 Concept Identifier

The purpose of the Concept Identifier component is to identify specified feature type from the input question and link it to corresponding classes, depending on dataset ontologies. Again, this is easier understood using a simple example:

The component takes this sentence as input:

- *Which hospitals are there in Oxford?*

and outputs:

- *Which (hospitals)$_{dbpedia.org/resource/Hospital}$ are there in Oxford?*

From now on we will be talking about those features as **Concepts**.

3.1.3 Geospatial Relation Detector

The Geospatial Relation Detector component is responsible for identifying the geospatial relations of the concepts and instances within the input question. It achieves that by categorizing specific words and their synonyms into specific categories and then "translating" each of the words into specific spatial functions.

Specifically, there are 3 geospatial relation categories:

Table 1: Geospatial Relation Categories – Relation Examples

Category	**Geospatial Relation**
Topological Relations	Within, crosses, borders
Distance Relations	Near, at most, at least
Cardinal Direction Relations	North, East, South, West

Of course, there is no intuitive deterministic way to translate some relations. To alleviate this issue, the component takes the liberty to map them to more specific quantitative relations, using a dictionary-based approach. For example, the *near* relation could be mapped to a *within 1 km* spatial function depending on the context. Technical details, about the implementation of those features, are of no use to us in this project. We will refer to the output of this component as **Relation** from now on**.**

3.1.4 Property Identifier

In a few words, the Property Identifier component identifies attributes of entities within the input question. In order to answer questions like “Which mountains in England have height more than 1000 meters”, we would need to be able to identify the *height* property of the concept *mountain* and filter out everything else that does not satisfy the condition specified. A property could be anything from the height of a mountain and the length of rivers to the population of villages and cities. The component achieves that, using lookup tables, containing attributes that a certain feature might have. For example, a “Mountain” feature contains the “height”, “elevation”, or “parent peak” feature, and so on. These attributes are then linked to the entity’s fields in the dataset and returned.

3.1.5 Query Generator

As the name implies, this component generates queries using the output of every other component before it. These queries will later be executed by the Strabon Query Executor to get the actual answer to our question by the GeoQA system. Without going into much detail, the component uses predefined query templates, given researched patterns that appear in most questions. such as CRI, IRI, CRIRI, or PCRI where I stands for Instance, C for Concept, R for relation, and P for property.

The component produces multiple queries, using all combinations of outputs from the previous components, and ranks them by relevance score. Notably, the queries are in SPARQL/GeoSPARQL. This language is the standard query language for linked data.

3.2 Communication

As explained earlier, we communicate with the GeoQA system, meaning with each of the Qanary components, and the Query Executor, via HTTP APIs. Each of the components is a program that is hosted on a server and provides APIs on specific endpoints and ports on that server. Each component category can contain a multitude of components implementing different ways to achieve the same or similar results, as long as the input and the output abide by a predetermined protocol.

In order to pose a question and extract the results from the GeoQA system we will need to follow the procedure below without going into technical detail:

1. Inform the Qanary Pipeline process of our question and the components we choose to use to answer that question. This outputs an entity, that contains information that the components need to communicate with each other when answering our specific question.
2. Using the **graph** we pose the question to each of the components
3. We extract the output of each of the components
4. Using the Query Generator’s output, we choose a generated query and ask the Strabon Query Executor to execute it for us.
5. We retrieve the results.

How we actually and technically communicate with the GeoQA system, how we extract, process, and reuse information is beyond the scope of this section and is part of many design and implementation decisions in this project as we will be discussing in the two main sections (5, 6) of this thesis.

3.3 Limitations

As with every system, GeoQA has its limitations. More specifically, while experimenting with the tools that were initially created to test and demonstrate the system, we noticed that it was taking a long time to answer questions, especially more complex ones, with multiple properties, concepts relating to one another, and so on. It is in our interest, therefore, to do everything in our power, to decrease the response time of our application. We will go over the methods we used in the design section.

Moreover, it should be noted, that at the time of writing, the system does not understand every possible way of posing a given question. As its official title suggests, it is *a template-based question answering system*, implying that the questions it's able to answer abide by some templates. Those templates are being expanded upon every day, but we should be aware of the way we pose questions to the system so that it can understand and answer them in the way we expect it to.

Now that we've gotten a glimpse of the system we're dealing with, let's do an overview of the application we created.

4. THE APPLICATION

In this section, we will show an overview of how to use the application created for the purposes of this thesis, before we go into any design or implementation details.

4.1 Asking a Question – Home Page

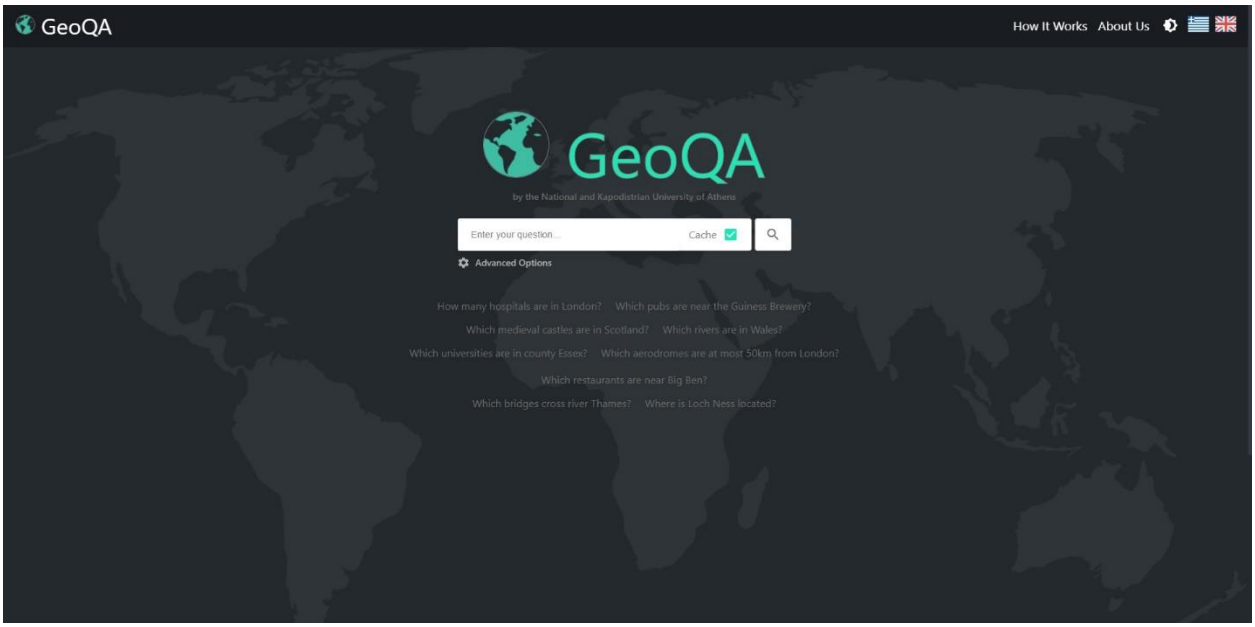

Figure 3: The Home Page

A very simple interface. The user can pose their question using the input field in the center. They can also click on one of the example questions below the input field to ask it.

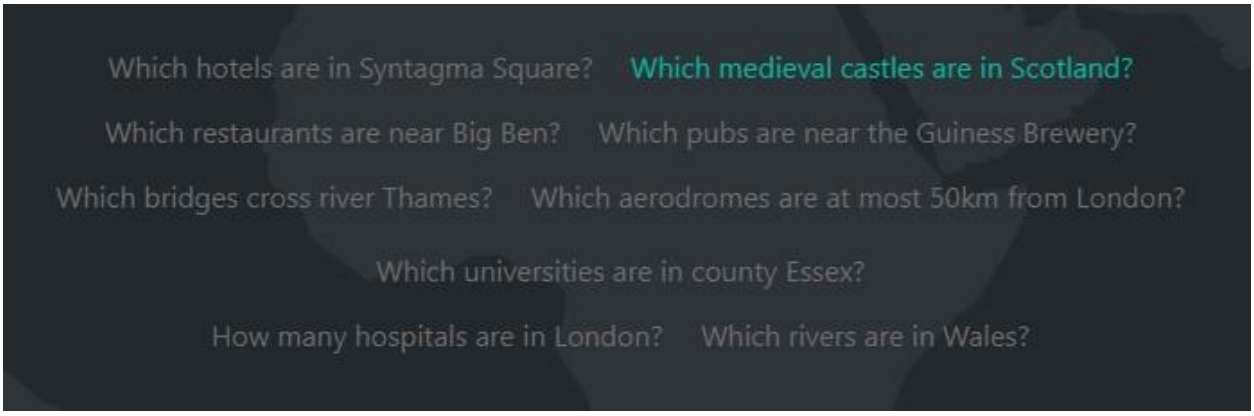

Figure 4: Example Questions

In the top-right corner of the window, the user can choose which display language they prefer, and also change the theme. There are 2 themes: Light and Dark (default).

We will go over what the “Cache” checkbox and “Advanced Options” button do later.

4.2 Displaying the Answer – Answer Page

Let’s test the application by asking it a question. “Which medieval castles are in Scotland?”

The info panel shows information about the results displayed, as well as a score, implying how confident the GeoQA system is on the results it's returning.

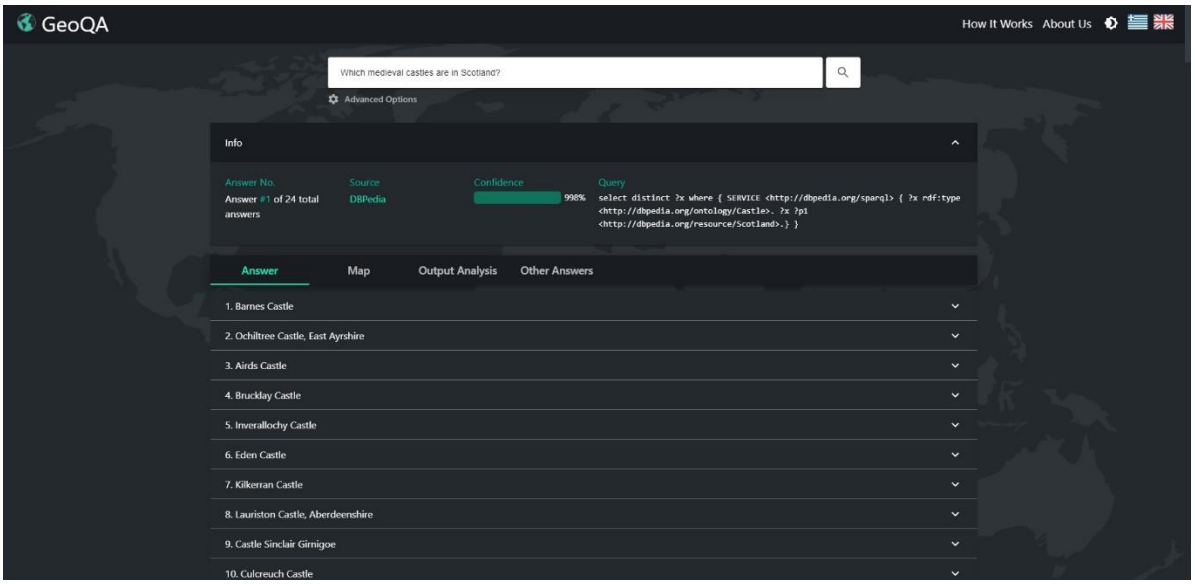

Figure 5: Answer Page - List

4.2.1 Answer List

This tab shows all named results in a list. Pressing an item will expand it, and reveal more information on that specific result, as well as give a link to an external wiki page if that exists

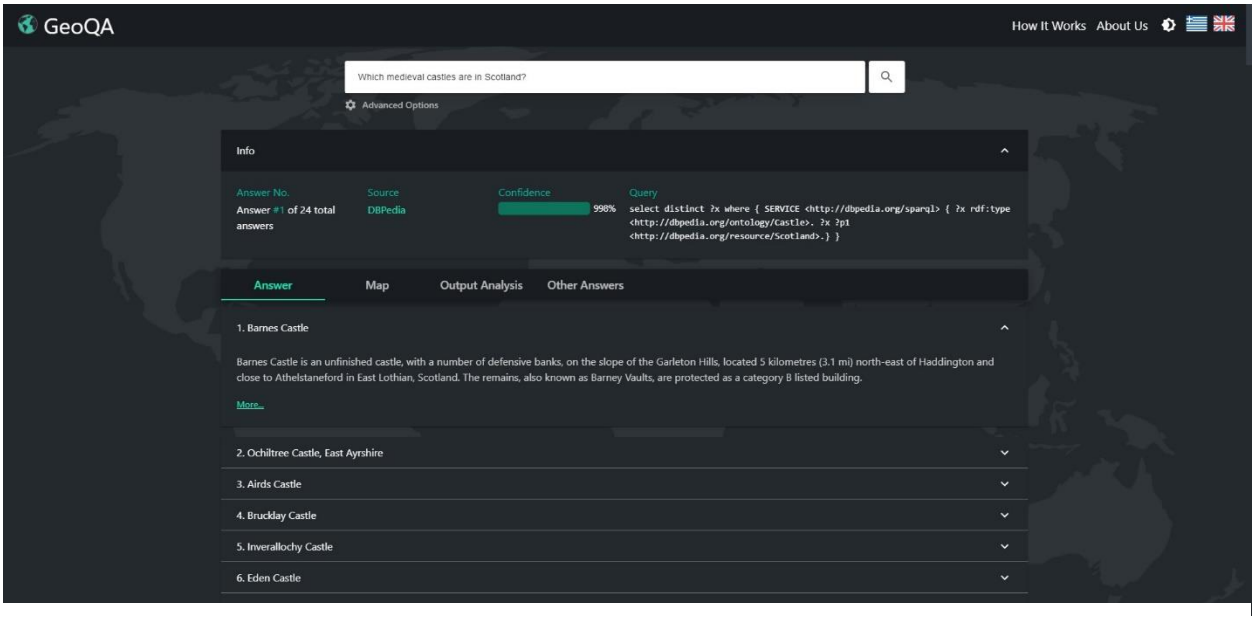

Figure 6: Answer Page - List Collapsed

4.2.2 Map

This tab shows a map with every result plotted onto it. Hovering over any marker will reveal the name of the result as well as other relevant information that may exist.

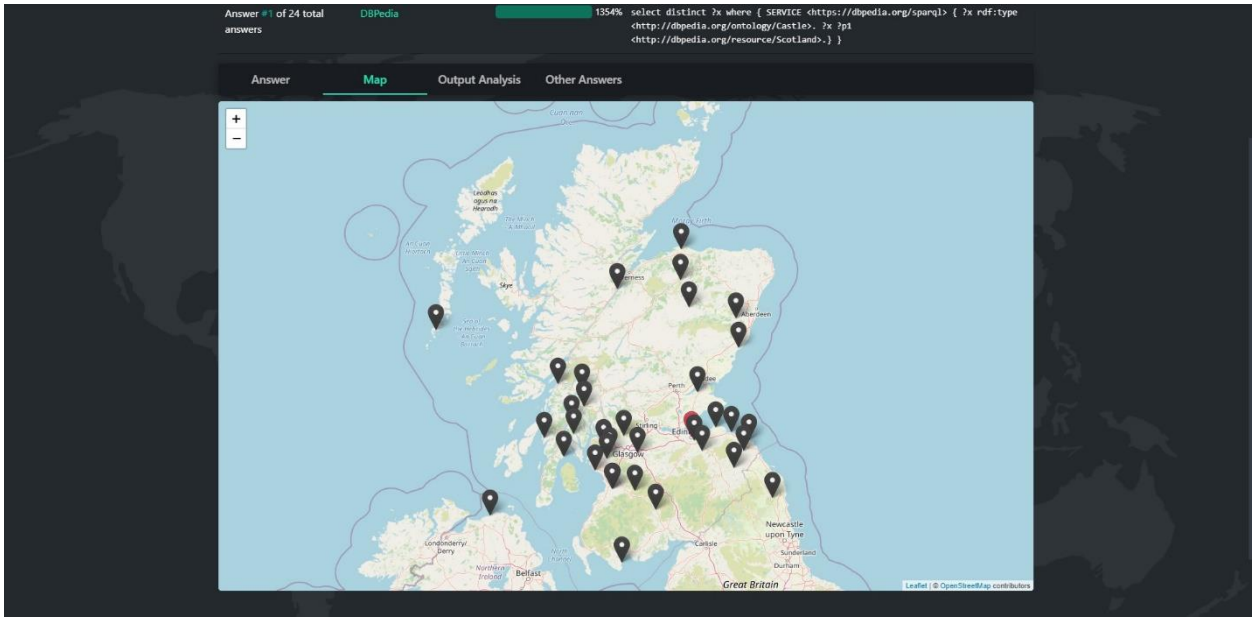

Figure 7: Answer Page – Map

The map also plots **the instance** as interpreted by the GeoQA system if the application finds coordinate information on it with a red color to differentiate it from all other results.

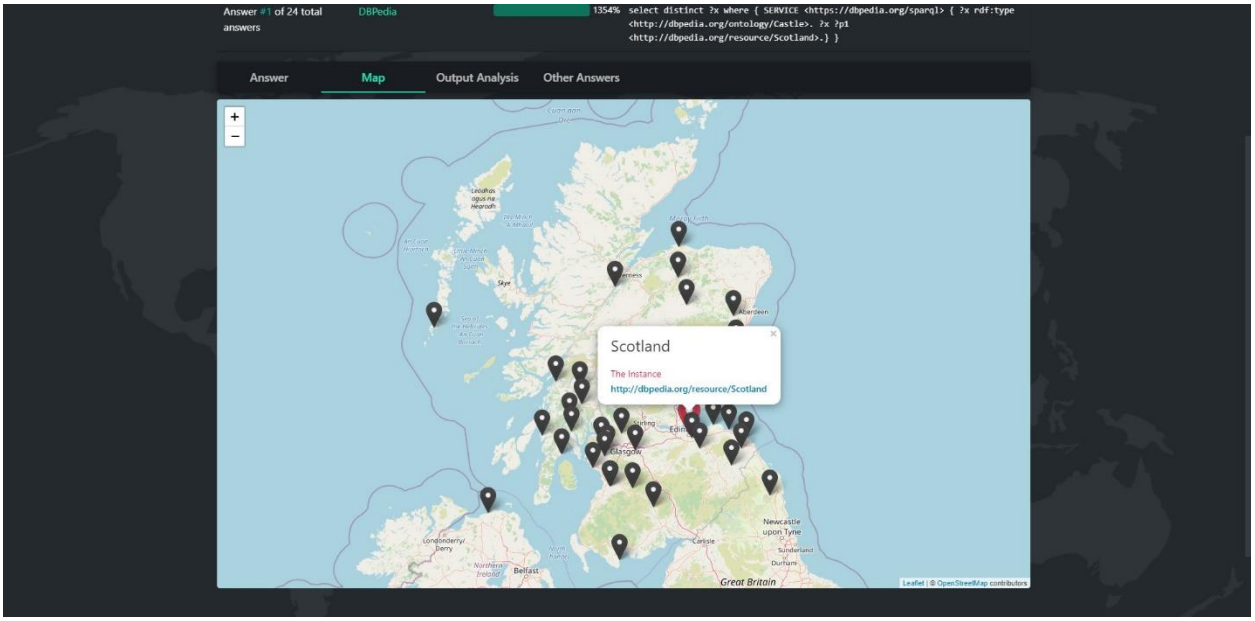

Figure 8: Answer Page - Map – Instance

If the question is a “near” or “within” question, a.k.a. the **relation** is “distance” or “within”, the map will also plot a radius around the instance.

4.2.3 Output Analysis

This tab displays every picked component's output, as well as which of them is selected in the current Query Generator query.

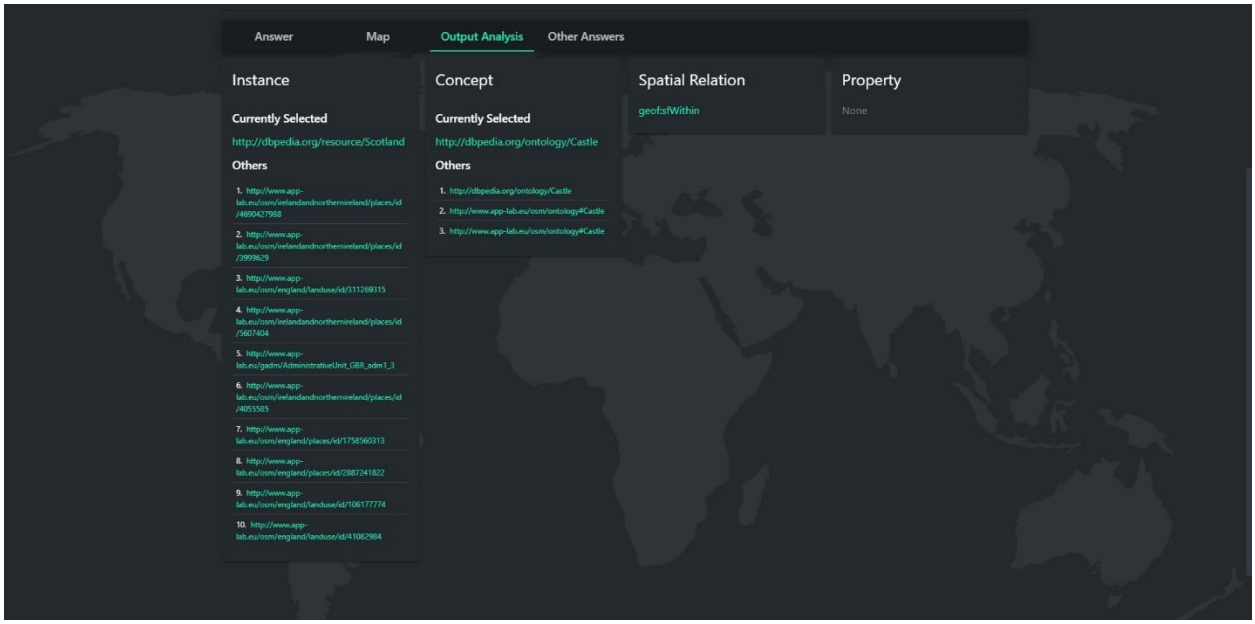

Figure 9: Answer Page - Output Analysis

Pressing any of the links will direct the user to that specific resource, or explanation.

4.2.4 Other Answers

This tab displays all queries returned from the Query Generator component except the one in use currently. Next to each query is how confident the GeoQA system is for that query. Pressing any of the "other answers" will attempt to answer the question using that specific query.

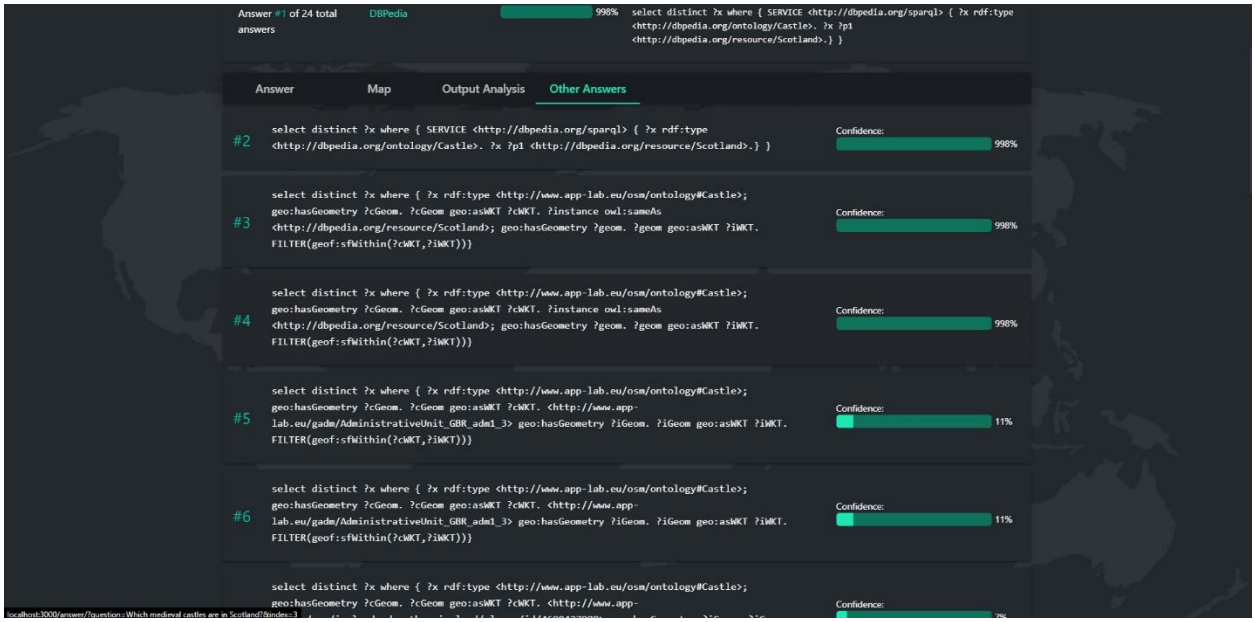

Figure 10: Answer Page - Other Answers

4.3 Customizing the Pipeline – Options

By pressing “Advanced Options” in the Home or Answer pages, a smaller window will appear. This interface is what allows the user to customize the way the GeoQA system will answer their question.

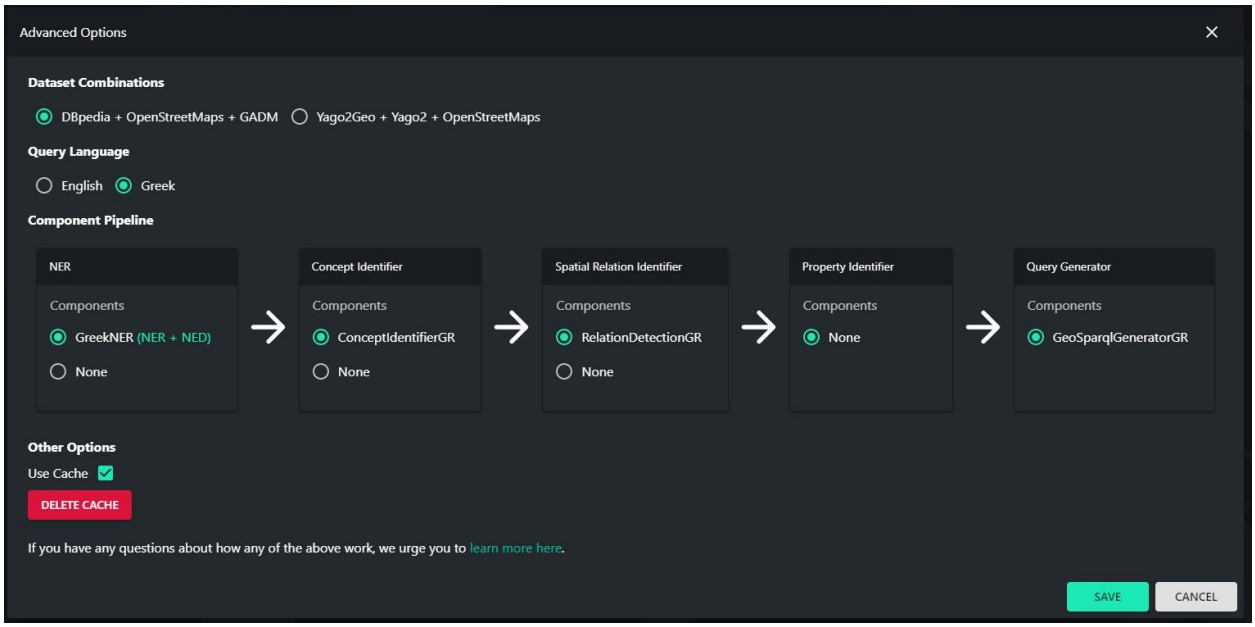

Figure 11: Options Interface

The user can pick which dataset combination they want to use, in what language they want to ask their question, and most importantly, which components they want to use to answer that question sorted by component category. Changing the dataset combination or the query language will result in a change to available components. The “Use Cache” checkbox toggles the cache mechanism which is being talked about in more technical sections. The “DELETE CACHE” button allows the administrator to delete the cache. Credentials are necessary to use that functionality. Pressing “CANCEL” discards all changes.

This concludes the application’s tutorial.

5. FRAMEWORKS AND DESIGN CHOICES

The goal of the GeoQA Interface project is to create a **user-friendly** way to use the GeoQA system. What does user-friendly mean in this case? Who is the user we need to be friendly towards and what does this mean for our design and implementation decisions? The answer to those questions is what this section discusses.

Let's discuss the former question. *Who is going to use this tool and its interface?* The application is mostly a demonstration and output analysis tool for the GeoQA system. This means, that the user we should be primarily designing for is the researcher, developer, and/or system maintainer. This doesn't mean that the user interface should be crowded and difficult to understand. It should be simple and allow **any** user to try it, regardless of his familiarity with the matter.

Moreover, the GeoQA system consists of a multitude of components and sub-components, databases, query executors, dictionaries, the combinations of which we want to be able to customize and interchange. Therefore, we need a fully customizable and abstracted system, that handles all possible combinations of component choices. We need an interface, that not only displays the system's output but also provides the user with a simple way to change its configuration. It should be able to easily switch from dataset *a* to dataset *b*, or component pipeline *c* to component pipeline *d*.

5.1 Why a web application?

A web application is by definition an abstracted modular application. As explained in the first section, it is comprised of the *front-end* and the *back-end*. In a few words, the front-end handles the actual user interface – what the user sees – while the back-end does the dirty work and provides the front-end with clean data and information.

More specifically, our back-end will implement an API (Application Programming Interface), that our front-end will *"connect"* to. That means, that the back-end is free to act independently, as long as the communication between the two parts remains consistent. For example, we could say that our back-end will act as an abstraction to the GeoQA system. It will handle all communication with the system, and only return clean and useful information to the front-end. This will be discussed further in later sections. This solves our abstraction problem pretty well.

Another very useful feature that a web application provides, is that it can be accessed from anywhere, as long as there is an internet connection. There is no need for any installation process, disk space, or local computer resources. The internet connection restriction is not a restriction in our case. The GeoQA system and all of its components also implement an API that resides on the web. We wouldn't be able to connect to that system without an internet connection. Therefore, the restriction is part of the initial hypothesis.

Lastly, at the time of writing, the web development community is the most active developer community out there. That means, that there is a plethora of development frameworks, libraries, quality of life tools, and sources, that make the development of this application faster and of higher quality. The choice and use of said frameworks and tools are being discussed in the following sections.

5.2 Design Philosophy and Modularization

Generally, we modularize the application as follows:

1. The Administration Platform
2. Back-end & API
3. Front-end

Other than the above modules, there exists another important development field. *Deployment.*

Below are listed the frameworks used per development field:

Table 2: Frameworks Used

Development Field	Framework Used
Back-end	Node.js
Front-end	React.js + Material UI
Administration Platform	Node.js + React.js + Material UI
Deployment	Docker

One of the reasons we picked Node.js and React.js for this web application, is that they use the same programming language. Javascript. While there may exist better or more efficient tools to create back-end systems, we found it easier to have to work with one single language, so there is no conflict between data-types in inter-application communication. The GeoQA system and the way it extracts information is complicated enough, to have to “translate” it multiple times between the different stages of communication. The use of a single programming language solves that problem.

Other than the above note, each framework will be discussed in its development field’s corresponding section.

5.2.1 Back-end

The GeoQA system is a complicated one. It consists of many services and sub-services that may or may not be connected. We need a way to centralize communication with those services. We want to ask our geospatial question and get one single output, containing all relevant answer information.

The Back-end acts as the middle man between the Front-end and the GeoQA system. Its primary job is to handle all communication with each of the components and knowledge bases. It implements a REST API, that responds to requests coming from the Front-end.

Firstly, it passes on component information from the administration platform to the front-end. Details on the nature of that information and the administration platform are discussed in sections 5.2.3 and 6.1.

Secondly, and most importantly, it handles the question requests. It first gets the actual question string, and the components the user wants to use to answer the question from the Front-end. Afterwards, it initializes a pipeline using that information (more details on the pipeline in section 6.2.1). It then starts a procedure, where all components are being asked the question in sequence while logging their output. Any errors will cause the procedure to stop and inform the Front-end with a detailed error message. If everything goes as expected, it will return each component’s output to the Front-end.

As explained in section 2, the above procedure is not enough to actually answer the question. In order to get the actual relevant output from the components, we need to look at the Query Generator component's output. It will usually be a SPARQL query. We decided that the query extraction from the Query Generator's output is not something the Back-end should do. The Back-end's job is to be given a query and execute that query alone. The reasoning behind this decision is simple. Executing such queries might take really, or even unacceptably long to finish, as they may or may not contain computationally expensive calculations. Therefore, we decide, that the Front-end, having full knowledge of every single component's output, should be given the task to decide which query it wants to run, or if it wants to run all or even none of them.

Continuing with the question procedure, the Back-end receives an output query, executes it, and returns the output to the Front-end. This output, as explained in section 2 could be of a multitude of types. At this point, we let the Front-end discern the type and decide how to handle (display) the output based on that. In the case that the output is a URI or a list of URIs, the Back-end provides functions that query the knowledge bases those URIs come from. For example, if a URI comes from DBpedia we provide a function that queries the DBpedia SPARQL endpoint, and so on. We can ask for specific information such as Name, Abstract, Map Coordinates, and such.

It is important to note here, that any of the above procedures might take a long time to finish, and there's not something we can do directly. What we can do, though, is to implement a caching mechanism for every single one of them. Specifically, it would be wise to cache:

1. **Question, Components List à Components Output**, for the component questioning procedure
2. **Query à Result**, for the Query Generator's output query execution procedure

We decided not to cache the knowledge base's output, for the simple reason that any of a specific resource's information could change at any given moment.

Lastly, the Back-end provides the Front-end with some example questions for demonstrative purposes. These questions are part of the GeoQuestion201 benchmark and are questions that the system should potentially be able to answer. As a side note, these were the questions that we used to test and debug our application.

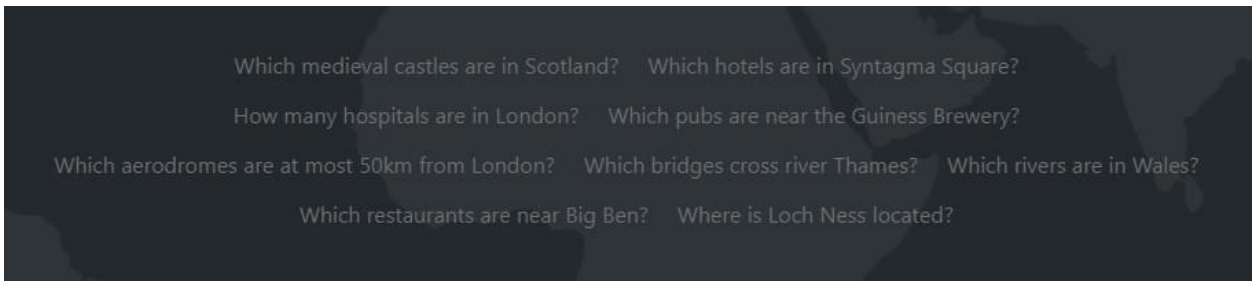

Figure 12: Example Questions

5.2.1.1 Frameworks and Technologies

As shown earlier we decided to use Node.js to develop the Back-end. Other than the same programming language argument that has been already discussed, other reasons are versatility, easiness of use, and the fact that Node.js has a very active and resourceful developer community, that provides solutions and tools for every problem a developer might face. It's very easy to build back-end servers and APIs with Node.js and it can handle all kinds of applications, from the simplest of web-apps to high-demand enterprise and commercial applications. The fact that Node.js is single-threaded poses no problem

to our application since it is not doing any serious and resource-intensive computations, but rather it processes information and handles communication between 2 separate systems.

For more information on Node.js, visit the Node.js website [3]

5.2.2 Front-end

The Front-end is the user interface. Its job is to provide the user with a way to pose questions, customize the pipeline, dataset, and input language that will be used to answer that question, and display the results. The idea behind the Front-end's design is simple. As noted earlier, our users can be both the common, *inexperienced* user, as well as the researcher, the programmer, or what we would call an *advanced* user. Our task is to make this procedure as simple as possible but also versatile, in terms of customizability and content.

The questions that the GeoQA system answers are *geospatial* questions. For example:

- Which medieval castles are in Scotland?
- Where is Loch Ness located?

As logic would suggest, we need to provide the user with not only the answer to their question in a verbose way (by text), e.g.,

- Which hotels are near Syntagma Square?
    - Grande Bretagne
    - Hilton Athens
    - …

Something like this would not be useful to our template user as we have defined it.

Therefore, we also need to:

- Provide descriptions of the results (places) or any other relevant information
- Plot the results on a map with markers
- Show each component's output and analyze it, depending on which component was chosen
- Plot any relational information on the map, e.g., a radius that defines "near", or a marker on the coordinates of Syntagma square in the above example question
- Show which Query Generator query was used, its score, and any other queries generated, while allowing the user to select which query they want to use.
- The source (knowledge base) from which our results are coming (DBpedia, Yago, OSM)

Before doing all of the above, the Front-end must first receive all of the information from the Back-end. Let's go over the generic use-case scenario.

5.2.2.1 Use Case Scenario

At first, the user asks a question. At this point, he can go into the options and select the dataset, language, and component pipeline. There is a default combination, of course, to cover the common user's needs for out-of-the-box functionality. The Front-end then poses the question to the Back-end and it returns the components' output as explained in section 5.2.1.

Afterwards, the Front-end parses the Query Generator's output (note that this component is picked by default and can't be unpicked), sorts all the results by score, and passes the best one to the Back-end for execution. We do not want to execute all queries since it is a very time-consuming process. The Back-end responds with the results.

Now, as explained earlier, the results can be of multiple types, so the Front-end must discern the type and potentially the source (if the type is a URI) of the results, and choose what to do with them. At this time, it also processes and combines all information from the query that was used, each components' output, the result type, and length into an entity we will from now on call **Reasoning**. Concurrently, it separates the results into **Main** and **Other** results, in order to show all other queries generated by the Query Generator, if any.

The Front-end now displays the results differently, depending on the type of the result, and its source. The main results info, component output analysis info, and other results info will be displayed regardless of what type the results are. For example, if a result is of type Boolean, e.g., for the question: "Is there a restaurant near Big Ben?", the user would see below Figure:

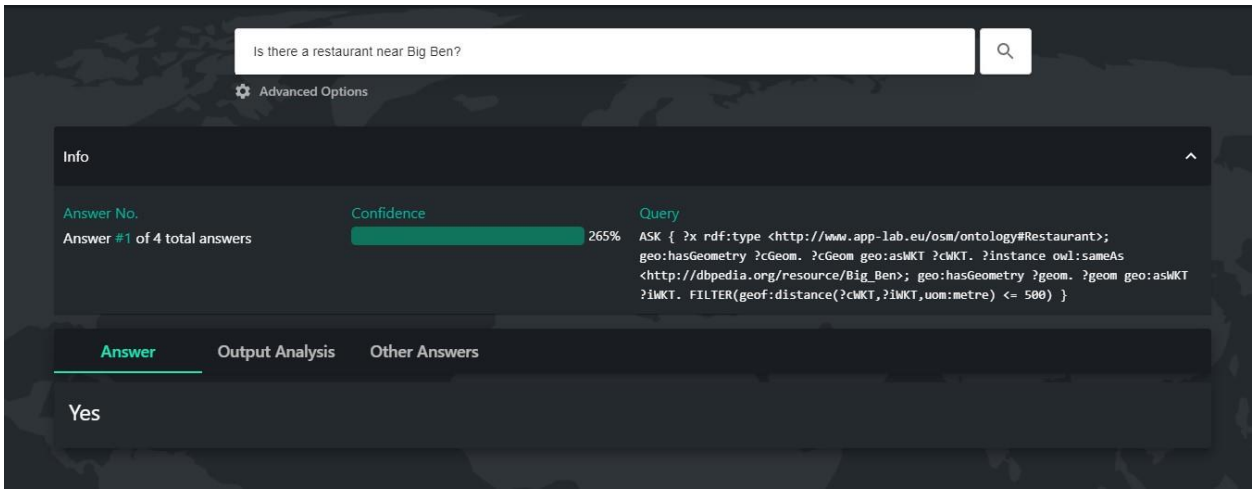

Figure 13: Boolean Results

If, on the other hand, the question was: "Which medieval castles are in Scotland" and therefore, the result type was URI and the source was DBpedia, the user would see these figures:

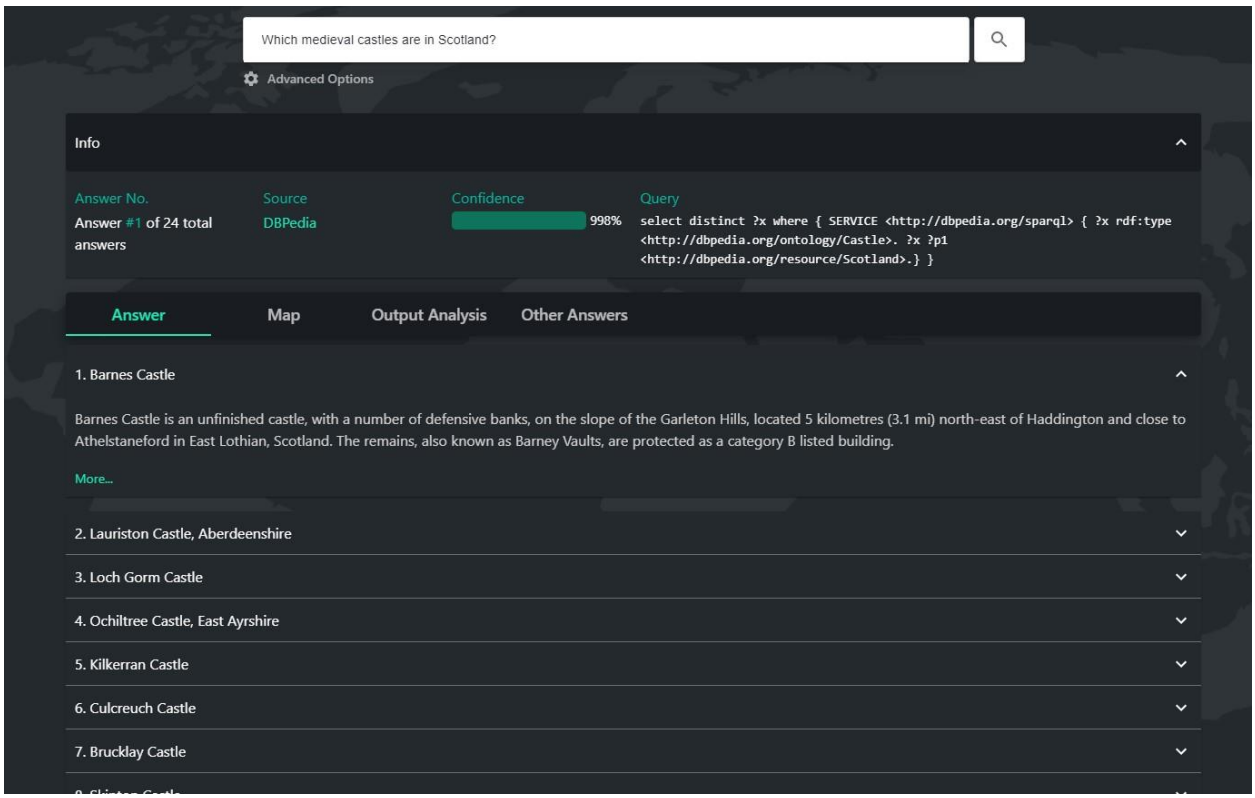

Figure 14: DBpedia Results List

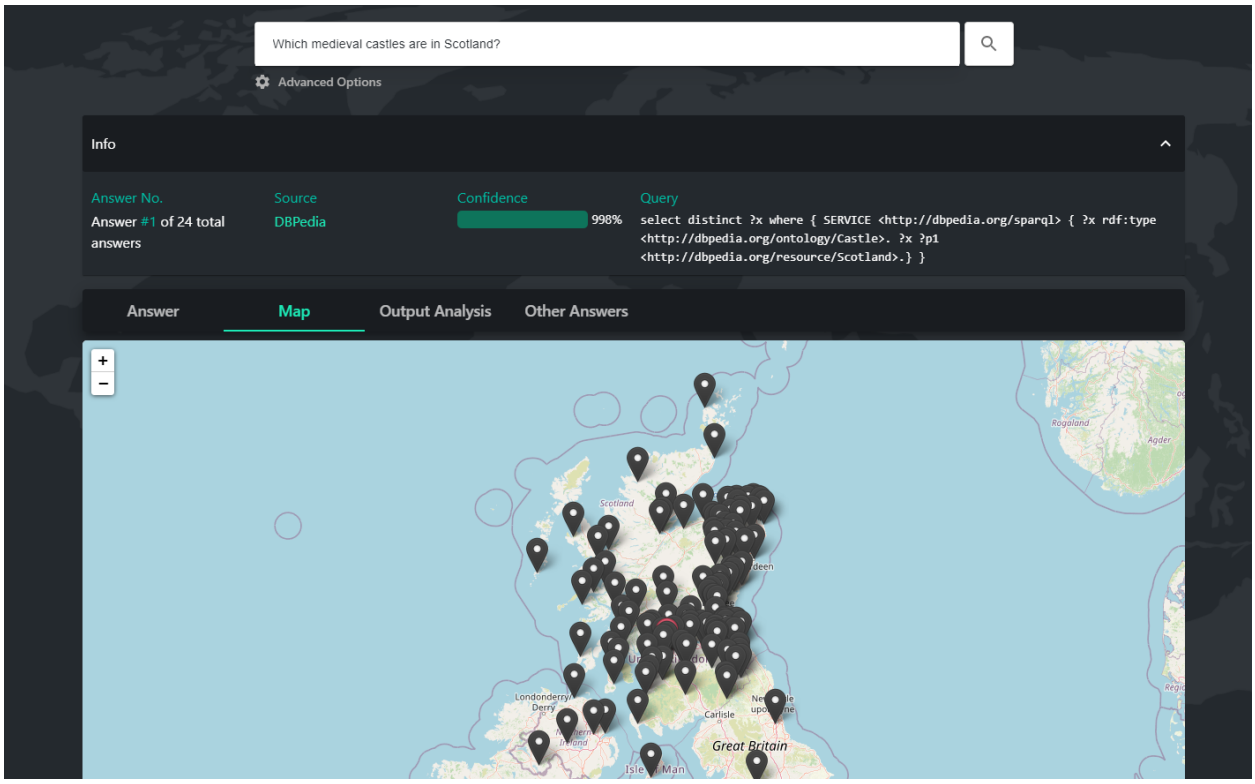

Figure 15: DBpedia Results Map

More information on every different result type and how it's handled is provided in section 6.3.1.

The user is now able to process the results as he pleases, use any external links, ask another question, use a different query to ask the same question, or change dataset and component pipeline combination to do either of the above.

5.2.2.2 Translation System

There is one more feature the application was tasked to provide. The GeoQA system plans to provide more and more languages as it gets expanded upon. Consequently, we want to provide the user with the respective display language. Since we don't and can't know every single language that the GeoQA system will potentially provide, it would be wise if we created a generalized dictionary system.

The solution we came up with, is a set of files, where each file corresponds to each language. Each file consists of a set of lemmata, which are [lemma name, translation in this language] pairs. For example:

Table 3: English Translation

Lemma Name	Translation in this language
AdvancedOptions	Advanced Options
ByUniversity	By the National and Kapodistrian University of Athens
Department	Department of Informatics and Telecommunications
DevelopedBy	Developed by

Table 4: Greek Translation

Lemma Name	Translation in this language
AdvancedOptions	Σύνθετες Ρυθμίσεις
ByUniversity	Από το Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Department	Τμήμα Πληροφορικής και Τηλεπικοινωνιών
DevelopedBy	Αναπτύχθηκε από

We then create a helper wrapper function that takes lemma- name, and language as its arguments and returns the translation in the given language. Using this solution, whenever we want to support another language, we just create a new dictionary file, that implements all given lemmata and translates them to our wanted language.

5.2.2.3 Frameworks and Technologies

We choose to use React.js to implement the Front-end. [4]. At the time of writing, React.js is one of the most if not the most popular Front-end library, and along with it comes a very active and passionate developer community.

In combination with React.js, we use the styling solution of Material UI. Material UI provides us with tested and well-designed UI components, like buttons, icons, panels, e.t.c., while also allowing for centralized theming. This makes development much less time-consuming and frustrating, allowing us to focus more on designing for our users, and thinking on a higher level, rather than reinventing the wheel.

5.2.3 The Administration Platform

The GeoQA components and the pipeline are separate java programs, that connect with each other and the rest of the GeoQA system as explained in section 3. These programs will need to be running somewhere to ask them questions. It would also be necessary to have a space where information about the components is stored. Information such as:

- Component name
- Component endpoint URL
- Component type (NER, NED, Relation Identifier, …)
- Component template question query for each component
- Component current status
- Dataset Combinations
- Available Languages
- Which components are used in which dataset combination and in what language

For those reasons, we create *the administration platform.* It has 3 primary jobs:

1. Run the pipeline program and every component
2. Store information about the components and datasets
3. Provide an API that passes the above information to the Back-end

It is a simple and separate web application, that also consists of a back-end and front-end. The back-end provides an API that responds to requests regarding:

- Component execution
- Information Retrieval

The front-end on the other hand at the time of writing is a simple interface that lists all available components and provides buttons that allow the administrator to execute or stop each component.

Figure 16: The Administration Platform Front-end

Information about the components is being stored in specific JSON files. These files act as the database for our application's back-end. More specific and technical information about the administration platform will be discussed in section 6.1.

We use the same technology for our back-end and front-end as the main application for consistency and simplicity reasons. Node.js provides plenty of process management tools (fork, exec, and the such), which makes for a perfect solution for the back-end, given that it also needs to implement the communication API. Moreover, as explained earlier, the fact that our systems are written in the same programming language is also very helpful and saves valuable development time.

Looking towards the future, we are planning on expanding this administration platform. It would be very helpful if it could do more than just running and stopping the components. It could, for example, allow uploading components, edit component information, change servers, start/stop/restart the main application, provide better component filtering and sorting, upload template questions, and more.

5.2.4 Overview

Now that we explained the core mechanics of our system, and before we go into the deployment section, we figured it would be helpful to provide a high-level overview of the system using a structure diagram. The diagram shows how information moves inside our system, the GeoQA system, and any knowledge bases we might be using for a given question.

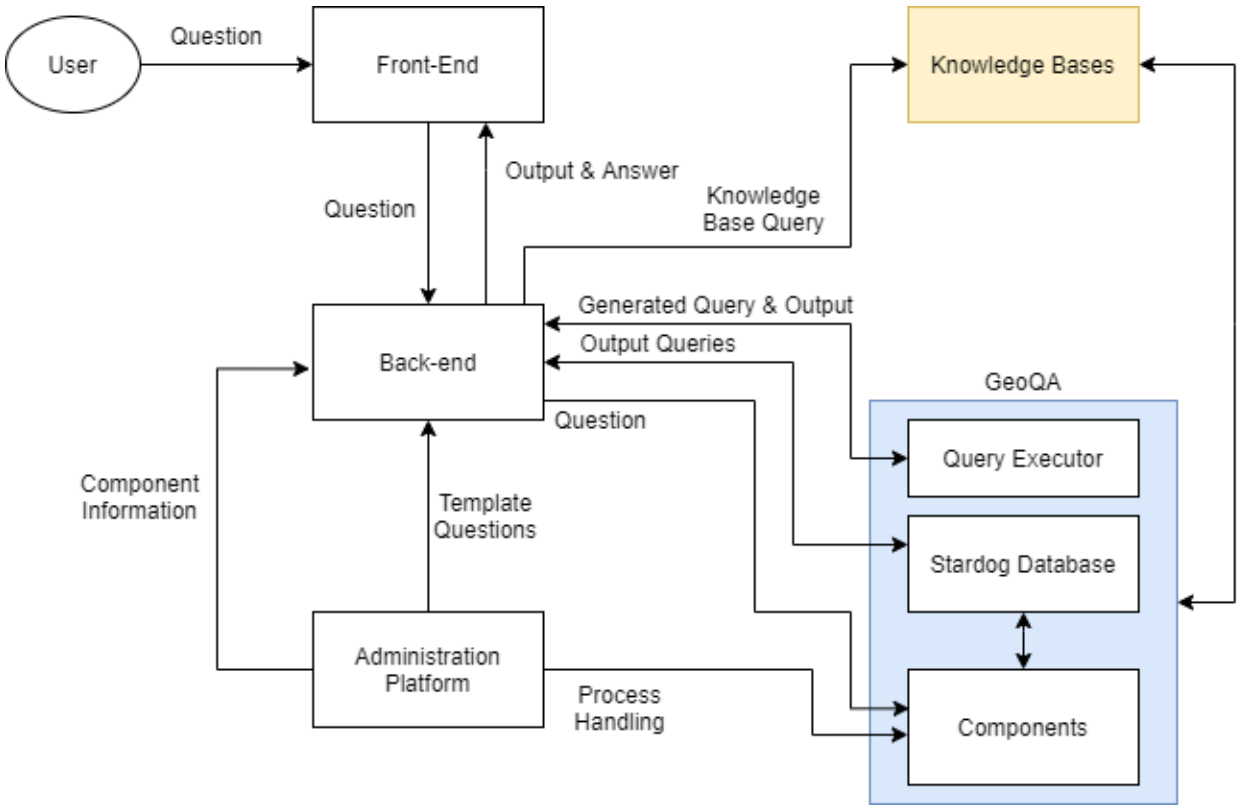

Figure 17: Structure Diagram

5.2.5 Deployment

The last development field is the *deployment* of the application. While there are no specific theoretical restrictions as to what technology we could use, it would be appreciated if the system was flexible and more importantly portable. We would also like for our system to run in a consistent environment to avoid any system configuration-specific problems that could arise.

We were given a specific set of servers to host our whole application. Those servers were specifically running Ubuntu 14.04. Unfortunately, this version of the operating system is old enough to not support the frameworks and libraries we wanted to develop our application with. This restriction along with the above discussion led us straight to *Docker*.

Docker provides the ability to package and run applications in isolated environments called containers. You can run multiple containers on a single host. You could compare containers to virtual machines, only that containers are much more light-weight than a virtual machine. For more information on how Docker works, visit the platform's documentation [5]. Ubuntu 14.04, with some configuration, was able to run Docker. That meant, that from this point on, we could run any environment we wanted on that server. Specifically, we needed to have 4 different environments for our 4 separate subsystems.

Table 5: Subsystem Environment Dependencies

Subsystem	Environment Dependencies
Administration Platform Back-end	Java (components), Python (Greek components), Node.js (back-end server)
Administration Platform Front-end	Node.js (Front-end React server)
Main Application Back-end	Node.js (Back-end server)
Main Application Front-end	Node.js (Front-end production building), Nginx (Front-end server)

Our whole application is now portable, and running in the same consistent environment anywhere we choose to host it, as long as the host has Docker installed. This is also much easier to do than having to install every single library and dependency every time we want to move servers or reinstall the application. Lastly, we have also created simple deployment scripts, that make moving and running the application, one line of terminal usage.

Below is the structure of our system in terms of deployment:

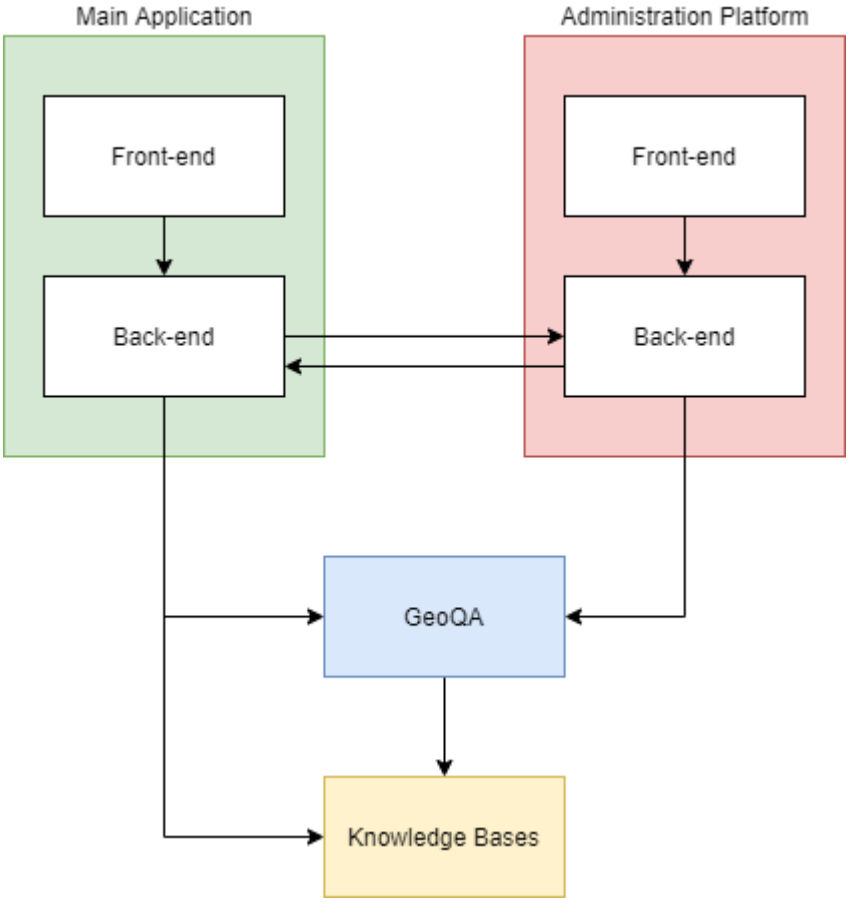

Figure 18: Deployment Structure

In the following section, we will discuss the technical and detailed implementation of the choices and procedures we explained in this section.

6. IMPLEMENTATION

In this section, we will discuss the technical details of the solutions introduced and discussed in the previous section. We will discuss how the Administration Platform runs each component and any of its dependencies, our caching scheme, how to use the REST API, and analyze our Front-end's component structure. Again, we will divide this section into 3 parts, each discussing a specific Development Field: **Administration Platform, Back-end, Front-end.**

6.1 The Administration Platform Implementation

Firstly, let's remember that the administration platform is an application that stores and passes information on all the available components, and also runs or stops them. We will go over the structure of the information stored, as well as how the platform runs, stops, and manages the components and any other programs it needs to run.

The Administration Platform's front-end is pretty straightforward and does not need further explanation, other than what was discussed in section 5.2.3.

6.1.1 Information Structure

There are 3 information entities that we need to discuss.

1. Dataset Combination
2. Query Language
3. Component Category (or Type)
4. Component

The stored information is structured and used in this exact order. Specifically:

1. A *Dataset Combination* supports some *Query Languages*
2. A *Query Language* contains some *Component Categories*
3. Each *Component Category* contains specific *Components*

We have 2 Dataset Combinations at the time of writing:

- DBpedia + OSM + GADM (from now on called *DBpedia*)
- Yago2Geo + Yago2 + OSM (from now on called *Yago*)

We have 7 distinct Component Categories as discussed in section 3.1:

- NER + NED
- NER
- NED
- Concept Identifier
- Spatial Relation Identifier
- Property Identifier
- Query Generator

For each category in that language, in that dataset combination, we keep a set of components. The categories are the same for each dataset combination and language, but the specific components are different in each case. We structure our data this way, in order to help the user decide which components they want to use, given their dataset

combination and query language preference. More information on that implementation in section 6.3.

As explained, each component is a java program that provides an API to query it. Let's go over the information stored for each component.

- Name
- Category
- Filename (.jar)
- Port
- Endpoint suffix

```
    "TagMeDisambiguate": {
        "port": 5555,
        "suffix": "/annotatequestion",
        "name": "TagMeDisambiguate",
        "jar": "qanary_component-TagMeDisambiguate-0.0.1.jar",
        "type": "NER + NED"
    },
    "AGDISTIS_Yago": {
        "port": 7004,
        "suffix": "/annotatequestion",
        "name": "AGDISTIS_Yago",
        "jar": "qanary_component-AGDISTIS-0.0.1_yago.jar",
        "type": "NED"
    }
```

Figure 19: Component Information Example

In order to query the components, we need to store the actual SPARQL query strings. Thankfully, we don't need to store distinct queries for each component. The queries are only distinct between different component categories. Therefore, we store template queries based on each component category, that can change for each different question that we ask. As explained in section 3, each question is represented inside the GeoQA system by a graph. There is a specific place inside each query that this graph should be. For that reason, we use a specific substring identifier <!graph!> inside each template query at its correct position. The Back-end then should replace it with the question's corresponding graph. More information on that procedure in section 6.2.1

```
    "Concept Identifier": {
        "outputQuery": [
            "PREFIX qa: <http://www.wdaqua.eu/qa#>",
            "PREFIX oa: <http://www.w3.org/ns/openannotation/core/>",
            "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>",
            "SELECT ?start ?end ?uri",
            "FROM <!graph!>",
            "WHERE {",
            "    ?a a qa:AnnotationOfConcepts .",
            "    ?a oa:hasTarget [",
            "        a oa:SpecificResource; ",
            "        oa:hasSource    ?q; ",
            "        oa:hasSelector  [",
            "            a oa:TextPositionSelector ;",
            "            oa:start ?start ;",
            "            oa:end   ?end",
            "        ]",
            "    ] .",
            "    ?a oa:hasBody ?uri ;",
            "    oa:annotatedBy ?annotator ",
            "}",
            "ORDER BY ?start"
        ]
    },
```

Figure 20: Component Category Query Example

All of the above information can be passed on in many different forms to whomsoever might need it (e.g. the Back-end) using analogous REST API requests.

6.1.2 Running and Stopping the Components

Node.js provides us with a library that allows spawning and managing subprocesses. The *child_process* library [6]. We have 3 things that we need to run. The pipeline program, the components, and any other programs the components might depend on. At the time of writing, the only components that have any dependencies are the Greek language components. They depended on 2 python3 services, which should be running on the machine the components were running on.

We decided that we did not want the components to run all the time since they would potentially demand precious server resources. However, the pipeline program needs to run at all times, so that it can register new components and execute the initial queries for each question, regardless of what components are running at any given moment. The same goes for the dependencies. Therefore, we should run the pipeline and the dependencies on startup, while letting the components be run on demand using REST API requests.

We also store runtime information on what processes are being run at all times, while logging their output to specific log files for each one, and storing references to them.

Lastly, another important thing that needs to be mentioned, is the cleanup process. We decided, that it would be better if we peacefully exited any processes still running when

the administration platform stops for any reason. That way, we avoided any serious errors and log file corruption. Thus, we created a method that kills all running processes when the application exits, receives a SIGINT, SIGUSR1, SIGUSR2, or when an unhandled exception occurs.

6.2 The Back-end Implementation

As we have said plenty of times now, the Back-end acts as the middleman between the GeoQA system and our Front-end. Therefore, we need to implement the bridges to both those components. We have already gone over the use-case scenario and question procedure in sections 5.2.2 and 5.2.1 respectively, so we will not go over it in more detail. This section focuses mostly on technical details and implementation at a lower level.

6.2.1 Connection with the GeoQA system

Connecting and using the GeoQA system happens mainly using the REST APIs provided by it. We need to connect with:

1. The pipeline process for the initial query
2. Each component used in a given question's chosen component pipeline
3. The database containing the output of the components
4. The database containing the answer to the question, e.g. the results of the Query Generator component's generated query
5. Any knowledge bases that contain more specific information about the question's output.

All of the above implement REST APIs that we can use to achieve communication. As explained, a question is answered in the same order as the connections we're describing above, but not without extra steps or system interaction between them.

We bundle the first 3 connections in 1 procedure while handling the 4th and 5th connection as independent procedures, for the Front-end to invoke.

We created the Pipeline entity (class), an instance of which is initialized every time a question is asked. After initialization, - using the query method – the Pipeline instance queries the pipeline process. Using that query's output it asks every chosen component in sequence. Lastly, it queries the database for every chosen component's output. In order to achieve the last part, we will need to replace the output graph of our question – that is contained in the initial query's output - in the Component Category's Output Queries, which are explained in section 6.1.1.

Anything beyond that specific procedure, for example, sending the output back to the Front-end, is not handled by the Pipeline instance. We will talk about that, as well as caching that output, in the next section.

6.2.2 Connection with the Front-end - API

It's time to talk about receiving and sending information from and to the Front-end. We will list all available REST endpoints that the Front-end uses, what data they need, what they're used for, and what information they return, in order of use case scenario sequence:

1. ***/api/question***

   This endpoint is used by the Front-end when the user first asks a question. It receives the question and the components chosen by the user and initializes a pipeline using that information as explained in the previous section. It then queries

the pipeline and awaits **the output of every component**, which is what it also returns to the Front-end once everything's completed successfully.

2. ***/api/result***

   This component's purpose is to receive a query, generally an output query from the Query Generator component, and query the GeoQA database using that query. If the query has been executed successfully, its output will be returned to the Front-end. That output could be a list of URIs, a Boolean answer, e.t.c.

From now on, the endpoints are not used in sequence, but rather conditionally based on the output type (URI, Boolean, e.t.c.) and its source (OSM, DBpedia, e.t.c.). We have created template queries for each data source, similar to the queries created for the component categories, only that in these queries the URI is the variable.

3. ***/api/map***

   If the output type is URI and its source is a map-based knowledge-base like GADM or OSM or YAGO2geo, then the Front-end uses this endpoint to derive more useful data from that knowledge-base entry than just the plain URI. For example, it could return coordinates, a name, and more. It takes a URI as its input and returns the output of the query executed in the knowledge-base.

4. ***/api/dbpedia***

   If the output type is URI and its source is DBpedia, this endpoint is used to derive more information about the entry. Input and output types are the same as the ***/api/map*** endpoint.

5. ***/api/yago***

   Similar to ***/api/map*** and ***/api/dbpedia***, this endpoint is used when the output type is URI and the source is Yago. Input and output types are the same as the ***/api/map*** endpoint.

The questioning procedure ends here. The endpoints described below are general-purpose endpoints.

6. ***/api/components***

   This endpoint returns all available components, structured in the way that was described in section 6.1.1. It's used by the Front-end for simple information retrieval.

7. ***/api/questions***

   This endpoint returns a list of sample questions, that the Front-end displays on its front page.

The procedures implemented in endpoints 1 – 4, might take a long time to complete. Therefore we decided to implement a caching mechanism, to decrease the amount of time a user waits for his question to be answered.

6.2.3 Caching and Polling

We will first need to figure out what exactly is it that's taking too long to complete. Only then will we be able to think of a good system that solves or at least partly solves the delay.

Generally, the knowledge base querying is a fast procedure, since usually, the queries are simple SELECT queries. Moreover, big knowledge bases such as OSM and DBpedia offer optimized and scaled APIs. That also greatly decreases the time of service.

Therefore, we only need to optimize the parts of our procedure that specifically interact with the GeoQA components. The components themselves, the pipeline storing their output, and the database storing the question output.

We don't have any control over the above parts. We will need to come up with an external system, that decreases wait time. Obviously, we are going to implement caching mechanisms. Caching is all about reusing information that has been used recently or is being used very often. Let's think about what could be reused in our specific case.

We can assume that our application, in tandem with the GeoQA system is a deterministic system in a specific moderately big time period. When you ask the application a question in that time period, the answer will always be the same. The only reason, where it wouldn't be the same, is when a component stops running or is updated. That would indicate a cache reset.

More specifically, when you ask the application **a specific question** with **a specific set of components**, the output of said components will always be the same. Therefore, this output can be cached.

As we have explained before, after the output of the components is retrieved, we query the GeoQA database using the Query Generator's output query. But that query is also the same, for a specific question with a specific set of components. We can also assume that the GeoQA database is a deterministic system in that same time period. Therefore, the results of that query will also be the same and can thus be cached.

As for the knowledge base querying, we explained how they are fast enough that caching is not necessary. Additionally, we have no way of knowing whether a knowledge base has changed their data. So we figure it's safer to not implement caching for these procedures.

6.2.3.1 Caching procedure

When a user asks a question, the Back-end receives that request and starts querying the components. At that moment, it creates a cache entry that stores a "Pending" status and sends that "Pending" back to the Front-end. When the querying is completed, and the components send their output, the output is stored in that cache entry.

In case an error happens, the Back-end sets that cache entry to "Rejected" with a timestamp attached next to the status. The Back-end sends that "Rejected" status for 10 seconds until it tries to query the components again.

The Front-end then asks the Back-end every some amount of seconds until it gets a non-"Pending" or "Rejected" response. The amount of seconds, the Front-end waits to retry depends on the number of concurrent users. At the time of writing, since our users are not many, the Front-end waits 1 second before it retries the question.

The same procedure is followed for the answer results (**/api/result)** as well.

6.3 The Front-end Implementation

The Front-end is built using React.js. React.js apps are built with a component-based structure. One could argue that React.js programming is **declarative** programming rather than **imperative.** This is because React components are entities that have a state. Based on that state, the component's output changes. A React component declares logic that is executed on specific events (like loading, mounting, updating, unmounting, e.t.c.), or state changes, and then outputs text, input fields, and other React components. The components we have created for this application are what we're primarily discussing in this section.

But before we go into the component structure, let's talk about global values that we store in the browser's "localStorage" and use throughout the application and in different sessions.

1. Theme

   The application provides a dark and light theme. Based on this variable, the colors and general style of the application change.

2. Language

   As explained, we provide multiple languages for the application. We consider the functionality of this variable obvious.

3. Component Pipeline

   This is the most important of locally stored variables. This defines what components will be used when asking a question. It is modified using the Options component, which we will talk about later.

4. Dataset Combination and Query Language

   These variables define what dataset combination and query language as they were defined in section 6.1.1, that the user has selected. The set of available components to choose from depends on the value of these variables.

6.3.1 Component Structure

Below are shown the core components of the Front-end application, as well as how they are structured in relation to one another.

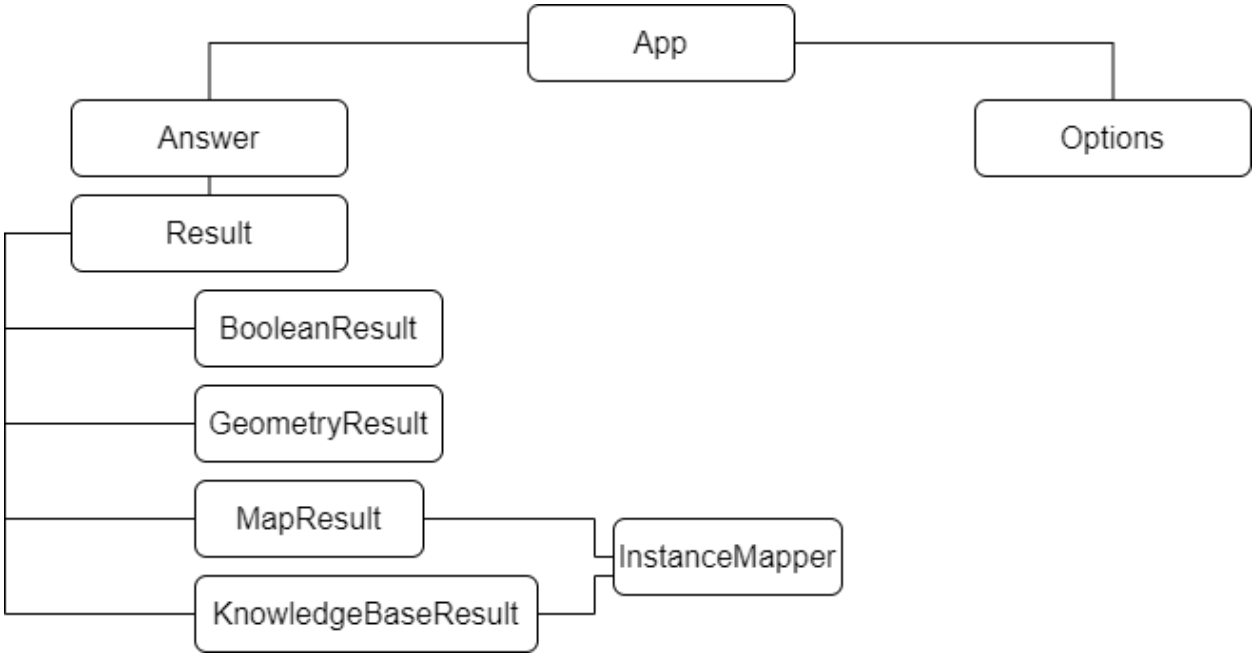

Figure 21: Front-end Component Structure

Let's go over what each of the components does:

The **App** component is the root of our application. Firstly, it retrieves locally stored values or initializes them if they do not yet exist. It renders the app's header and background image, while also providing routing. One of its routes is the **Ask** component. This component is the one that's actually rendering the **Answer** and **Options** components.

The **Ask** component also renders the input field for asking the question and the grid of template questions.

6.3.1.1 Result Displaying Components

The **Answer** component is the one responsible for asking the question to our Back-end. It retrieves the question for the URL and the component list from the locally stored variable and handles both questioning procedures:

1. Asking the components
2. Querying the GeoQA database with the Query Generator's query if any.

As explained in section 6.2.3, the component continuously asks the Back-end until it gets a non-"Pending" or "Rejected" response for each of the 2 procedures. Let's remember here that the Query Generator might return more than 1 query. Each of the queries is accompanied by a score when retrieved from the Query Generator's output. We sort those queries and choose to answer the one with the highest score. We do not use all queries, since that would take way too long. We do provide the user with the ability to choose any query they want though. This is achieved technically by using a URL parameter called "index", which defines the index of the query used in the sorted array. The way the user is able to select which query they would like to use is explained later.

When everything's completed it passes the results (even if there are no results) to the **Result** component. This component's responsibility is to first format the results into a more readable form. It then figures out the result's type. Those types are:

1. URI
2. Geometry
3. Number
4. Boolean

Simultaneously, the component extracts the **Reasoning** (see section 5.2.2) from the results. Now, depending on the type discerned, it renders different components as shown in the Component Structure Figure.

1. The **GeometryResult** component plots the geometry returned on a map.
2. The **NumberResult** and **BooleanResult** components simply display the Number or Boolean ("yes" or "no") answers.
3. In the case of URI, we need to further separate based on the type of source. We need to discern whether the URI refers to a "Map" knowledge-base like OSM or GADM, or from a full knowledge-base like DBpedia or Yago.

   When dealing with "Map" sources, we use the GeoQA database endpoint (the same we used in the **/api/result** endpoint) to gather more information on the results. Specifically, coordinates and names, if they exist. This is the job of the **MapResult** component.

   In the latter case, though we don't use a universal endpoint for every knowledge base available, the way we return information from those knowledge bases is structured, is the same every time. That's why we use a single component that queries different bases (using **/api/dbpedia** or **/api/yago/** for example) and displays the information.

Additionally, there was a need to display information that clearly shows how the system interpreted the question, and what difference that had in the answer. For example, let's say the question is: "Which restaurants are near Big Ben?". The GeoQA answers this

question in a very specific way. It sets a radius around **the instance** “Big Ben” (the instance) to satisfy the “near” **relation**. It then “returns” restaurants **(the concepts)** within that radius. It would be interesting if we could detect questions like this, and plot the coordinates of the instance, and display the radius around it as the GeoQA system decided. This is what the **InstanceMapper** component does. Firstly, it plots the instance on the map, if it exists. Secondly, it figures out the type of relation in the question. If that relation is a “distance” or “within” relation, it also creates a radius on the map around the instance.

6.3.1.2 The Options Component

The **Options** component’s job is to provide the user with a way to customize the way they want the GeoQA system to answer their question. It renders a dialog that lets the user pick:

1. The Dataset Combination
2. The Query Language
3. The Component Pipeline
4. Whether they want to use the cache (true by default)

The information on options 1-3 is retrieved from the administration platform as explained in section 6.1.1.

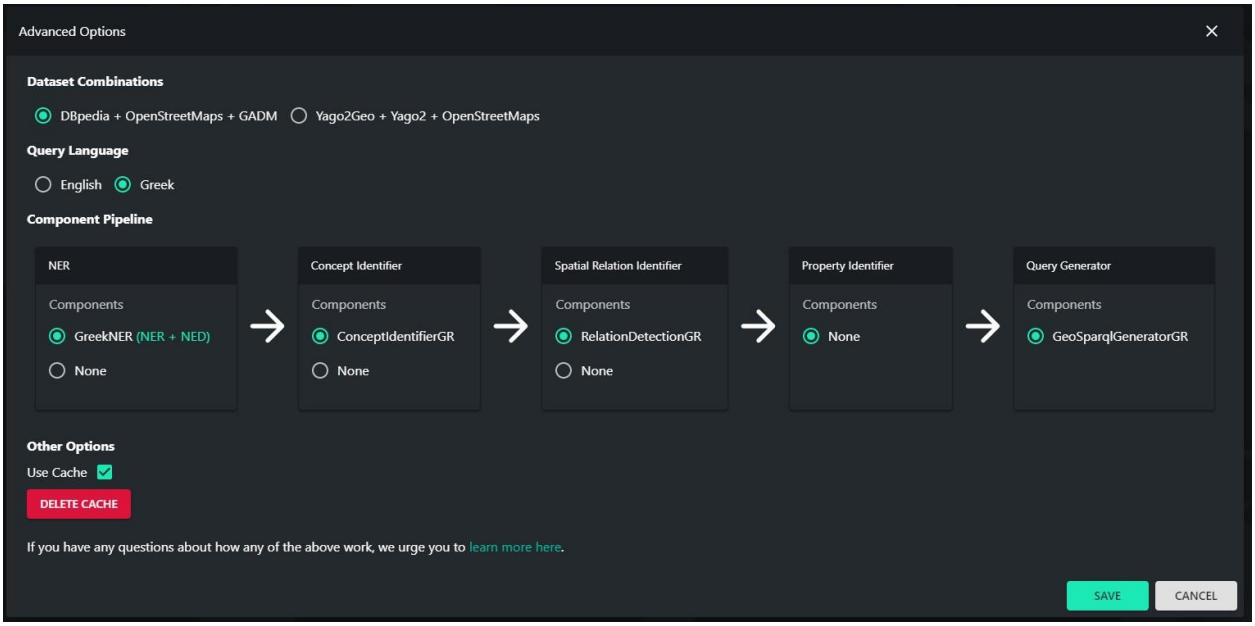

Figure 22: The Options Dialog

Whenever the dataset combination or query language changes, the pipeline of components changes as well to display the components available for that specific combination and query language as stored and retrieved from the administration platform.

When the user presses “SAVE”, all changes are locally stored in the user’s browser. That way, every time they use the system, it will remember their previous choices. If they press “CANCEL” all changes are discarded.

Lastly, the component allows the administrator to delete the cache if they find it necessary, using the “DELETE CACHE” button. Credentials are needed to access this functionality.

6.3.2 Routing

The Front-end implements client-side routing. Client-side routing is much faster than server-side routing for the simple reason, that it does not have to make requests to the server to render each page. It all happens via Javascript.

We decided to use routing to pass the question information to the **Answer** component. For example, if the question was “Which restaurants are near Big Ben?” the URL to that question would be “/answer/?question=Which restaurants are near Big Ben?”. That way, a user can share the answer to his specific question with another user. Additionally, it makes handling page reloads a lot easier since we don’t have to use local or session browser storage to store the question asked.

CONCLUSIONS

In this thesis, we have created a tool, that allows a user to ask a question in natural language and receive a visual answer and explanation. We focused, on making our interface friendly to different types of users we will get, and making the front-end of the GeoQA system versatile and extendible as the research progresses.

In the future, we plan to upgrade the Administration Platform, to an interface that makes it worthy of its name, and expand the main user interface to support more languages and temporal questions.

ABBREVIATIONS - ACRONYMS

API	Application Programming Interface
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
HTTP(S)	HyperText Transfer Protocol (Secure)
REST	Representational State Transfer
SPARQL	SPARQL Protocol and RDF Query Language

REFERENCES

[1] D. Punjani, S. Karan, A. Both, M. Koubarakis, I. Angelidis, K. Bereta, T. Beris, D. Bilidas, T. Ioannidis, N. Karalis, C. Lange, D.-A. Pantazi, C. Papaloukas, and G. Stamoulis, "Template-Based Question Answering over Linked Geospatial Data," in *12th Workshop on Geographic Information Retrieval*, Seattle WA USA, 2018.

[2] K. Singh, A. Sethupat Radhakrishna, A. Both, S. Shekapour, I. Lytra, R. Usbeck, A. Vyas, A. Khkimatullaev, D. Punjani, C. Lange, M. E. Vidal, J. Lehmann, and S. Auer, "Why Reinvent the Wheel: Let's Build Question Answering Systems Together," in *International Conference on World Wide Web, WWW*, Lyon France, 2018.

[3] "About | Node.js," [Online]. Available: https://nodejs.org/en/about/.

[4] "ReactJS Overview," [Online]. Available: https://www.tutorialspoint.com/reactjs/reactjs_overview.htm.

[5] "Docker Overview | Docker Documentation," [Online]. Available: https://docs.docker.com/get-started/overview/.

[6] "Child process | Node.js Documentation," [Online]. Available: https://nodejs.org/api/child_process.html.